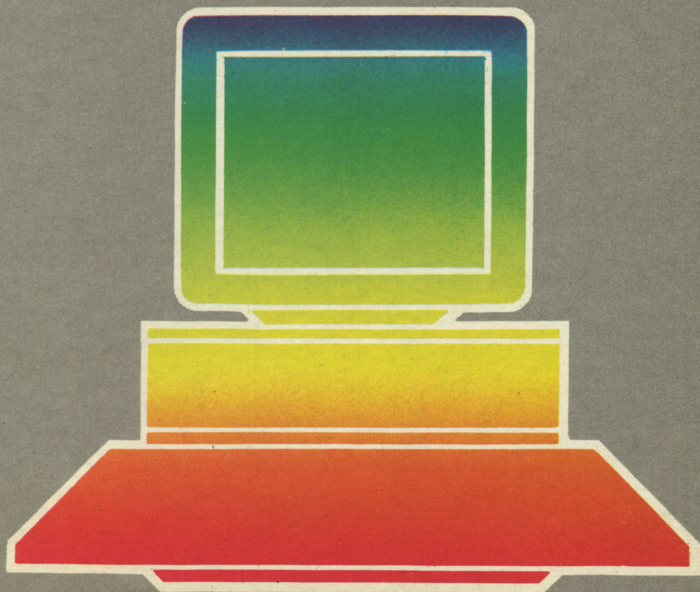
4

LE GUIDE DI BIT

il Linguaggio C

Friedman Wagner-Dobler

il Linguaggio C

Friedman Wagner-Dobler

Prima edizione: 1984
Titolo originale: C Language
Editore originale: PITMAN PUBLISHING LTD

TRADUZIONE: V. Cucchiaro
COPERTINA: Silvana Corbelli
FOTOCOMPOSIZIONE: Cencograf-Rotografica srl
P.zza S. Marco 1 - Milano - Tel. 655.20.13 - 655.51.45
STAMPA: Rotolito Lombarda S.p.A.

Contenuto del manuale

Note dell'editore 1
Come usare questo manuale 1
il "linguaggio C" 3
La libreria 52

Note dell'editore

DEC,PDP,VAX,RMS, e VMS appartengono alla Digital Equipment Corporation.
UNIX appartiene a A.T. & T. Bell Laboratories
CP/M apparțiene alla Digital Research Inc.
Microsoft e MS appartegnono alla Microsoft Corporation
Lattice è un marchio registrato della Lattice Inc. C 86 e Optimizing=C 86 appartengono alla Computer Innovation Inc.

Come usare questo manuale

Questo manuale descrive il linguaggio C, come è implementato su una diversa serie di processori e sistemi operativi. Oltre alla definizione del linguaggio di Kernighan e Ritchie*, dieci versioni del C language sono state consultate per i riferimenti all'utilizzo. Essi sono elencati insieme con le abbreviazioni usate per ognuno.

KR - Il linguaggio C definito da Brian W. Kernighan e Dennis M. Ritchie (UNIX)-1978
U7 - UNIX Version 7 C compiler-1979
CR - Cromenco C (8 bit Cromix/CDOS)-7 febbraio 1981
VAX - VAX 11 C Version 1.0 (RMS)-1982
CC - Control-CCC-86 (CP/M-86)-dicembre 1982
DR - Digital Research C Version 1.11 (CP/M-86)-ottobre 1983
CI - Computer Innovation Optimizing=C 86 2.0 (MS-DOS and CM/M-86) dicembre 1983
LAT - Lattice/Microsoft C Version 2.03 (MS-DOS) 1984
U5 - UNIX System V Release 2.0 C compiler-aprile 1984
Des - DeSMET C language Version 2.3 (MS-DOS and CM/M-86)-giugno 1984
MC - Microsoft C Version 3 (MS-DOS 2.1)

Occasionalmente, altri compilatori C sono stati presi in considerazione. In tutto il testo useremo il simbolo △ per indicare che una particolare caratteristica non è riportata nella serie sopra indicata.

* Brian W. Kernighan e Dennis M. Ritchie, The C Programming Language, Prentice Hall1978.

Notazione

Regole di sintassi

Le parole chiave e gli esempi in linguaggio C sono così rappresentati:

```
main() {/* sample program*/}
```

Le categorie sintattiche sono stampate in corsivo. Voci che appaiono su linee separate sono alternative mutualmente esclusive. Un <*elemento opzionale*> è indicato da [...]. I puntini di sospensione, '...' indicano ripetizioni opzionali, se preceduti da virgola, gli elementi ripetuti devono essere separati da virgola. Dettagli sui formati usati per descrivere la libreria del linguaggio C sono dati a pag. 65.

IL LINGUAGGIO C

Elementi

La serie dei caratteri del LINGUAGGIO C è dipendente dall'implementazione, ma una parte di essi è utilizzabile su tutti i sistemi. Tale parte è formata da caratteri grafici, lo spazio e caratteri speciali. Ognuno dei caratteri della serie della macchina può essere espresso con una sequenza di escape ma le sequenze di escape, che comprendono codici ottali e esadecimali non sono utilizzabile su tutti i calcolatori, eccetto che per il carattere NUL '\0'.

Caratteri grafici

Il termine 'caratteri grafici' verrà usato per i caratteri che sono utilizzabili all'interno degli identificatori. Ciò si riferisce a:

* Le lettere maiuscole A...Z
* Le lettere minuscole a...z
* I numeri decimali 0...9
* Il carattere di sottolineatura
* In alcuni casi il segno di dollaro $ △

Le lettere maiuscole e minuscole sono distinte tranne che nelle costanti esadecimali. I numeri decimali sono sempre in ordine ascendente e continuo.

Caratteri speciali

I seguenti caratteri speciali sono utilizzati dal linguaggio C

,	virgola	%	percentuale
.	punto	&	"e" commerciale
;	punto e virgola	^	accento circonflesso
:	due punti	*	asterisco
'	apostrofo	-	segno meno
"	virgolette	=	segno uguale
!	punto esclamativo	+	segno più
|	barra verticale	<>	parentesi angolate
/	slash	()	parentesi rotonde
\	backslash	[]	parentesi quadrate
~	tilde	{}	parentesi graffe
?	punto interrogativo	#	diesis

Spazio bianco

Lo spazio, i tab orizzontali, il newline, il ritorno carrello e l'avanzamento sono considerati tutti come spazi.

Alcune implementazioni considerano anche la fine del file (end of file) o i tab verticali come spazi. Il compilatore ignora gli spazi a meno che siano all'interno di una costante di caratteri o di una stringa; non è permesso porre degli spazi all'interno degli identificatori, di operatori multicarattere e di parole chiave. Alcune volte lo spazio è necessario per separare dei segni. Per esempio,

a+++b

significa **(a++)+(b)** dato che l'analizzatore cerca di raggruppare più segni possibili (p.e. '++')

a+ ++b

scritto usando lo spazio per separare i segni '+' e '++' è interpretato come **(a)+(++b).**

Sequenze di escape

Stringhe e costanti di carattere (p. 13) possono contenere sequenze di escape come le seguenti:

\n	new line	\d	caratteri con
\t	tab orizzontale	\dd	codice ottale
\b	back space	\ddd	d, dd,ddd
\r	return	\xh	caratteri con codici
\f	form feed	\xhh	esadecimali h,hh △
\'	apostrofo (carattere	\e	ASCII escape △
\"	virgolette (stringhe)	\v	tab verticale △
\\	backslash		

Il carattere backslash seguito da un new line serve come carattere di continuazione in stringhe ed istruzioni per il preprocessor. Per esempio:

△
```
char *string="is split over\
two lines";
```

Sfortunatamente, le limitazioni del compilatore spesso rendono questo uso non utilizzabile su tutti i calcolatori.

Parole chiave

Le parole chiave non possono essere usate come identificatori. Ma esse possono essere ridefinite nel preprocessor, per esempio:

#define void int definirà tutte le dichiarazioni **void** come **int.**

auto	**else**	**int**	**struct**
break	**enum**△	**long**	**switch**
case	**extern**	**register**	**typedef**
char	**float**	**return**	**union**
continue	**for**	**short**	**unsigned**
default	**goto**	**sizeof**	**void**△
do	**if**	**static**	**while**
double			

vari compilatori usano alcune parole chiave addizionali non standard; queste sono incluse nella tabella 8 a pag. 52

Commenti

I commenti sono delimitati dalle coppie di caratteri /* e */. Un commento può essere collocato ovunque potrebbe apparire lo spazio.

I commenti non possono essere collocati uno all'interno dell'altro, così **/* this /* is a */ bug */** è interpretato dal compilatore come **bug */**. Se parti di programma che contengono dei commenti non devono essere trattate dal compilatore, si dovrà usare le istruzioni del preprocessor **# if** o **# ifdef.** Cosi, se KNOCKIT è un simbolo non definito, la sequenza

```
# ifdef KNOCKIT
dummy () {/* deep freeze code */}
# endif
```

eviterà la compilazione della funzione **dummy.**

Identificatori

Gli identificatori sono utilizzati per assegnare nomi a variabili, funzioni e label. Un identificatore è costituito da una sequenza di caratteri grafici, ma con le seguenti restrizioni:

* il primo carattere non può essere un numero
* solo i primi otto caratteri hanno significato
* caratteri di sottolineatura all'inizio e alla fine possono creare conflitti con il nome di funzioni di libreria
* gli identificatori globali subiscono limitazioni dal linker: la

distinzione tra lettere maiuscole e minuscole può non essere più valida, sottolineature possono risultare illegali e solo i primi sette caratteri potrebbero risultare significativi.

Identificatori che si dovrebbero evitare sono elencati a pag. 52; la visibilità degli identificatori è discussa a pag. 39.

Esempi di identificatori corretti sono:

chkcrc	halfCnt	nophone
another	amber14	incount
nextR1	lumber	Knock
PrevEX	oddity9	identID

Costanti

Costanti intere

Le costanti intere possono essere date in notazione esadecimale, ottale o decimale. Le costanti esadecimali cominciano con **0x** o **0X,** le costanti ottali con **0** e le costanti decimali con i numeri dall'**1** al **9.** Ogni costante intera può essere preceduta da un segno meno, ma non dal segno più. Le costanti intere che eccedono la misura di un **int** sono convertite in **long.** Una costante intera seguita dalla lettera **l** o **L** è convertita in **long.** Non è possibile inserire degli spazi all'interno delle costanti intere.

Costanti di caratteri

Le costanti di caratteri sono singoli caratteri inclusi tra apostrofi. Le costanti multi-carattere non sono utilizzabili su tutti i calcolatori. Le sequenze di escape sono consentite. Le costanti di carattere sono di tipo **int.**

Costanti stringhe

Una stringa in C è un allineamento di caratteri terminante con il carattere NUL. Le stringhe sono costituite da zero o più caratteri racchiusi fra virgolette. Per le stringhe le sequenze di escape sono consentite. Le costanti generano un array statico di tipo **char.** Il carattere NUL è posto dopo l'ultimo carattere della stringa.

Cosi

```
char dog 1 [ ] = "wow";
char dog 2 [ ] = {'w', 'o', 'w', \ 0};
char dog 3 [4] = {'w', 'o', 'w', \ 0};
```

sono equivalenti. Quando usata in un espressione una stringa è trattata come l'indirizzo del suo primo carattere.

Stringhe scritte allo stesso modo possono o non possono essere allocate in posti distinti: alcuni compilatori assegneranno dodici caratteri per tre "wow", altri quattro. Dato che gli array **auto** non possono essere inizializzati,

```
main () /* wrong */
{
char dog 4 [] = "wow";
}
```

non è corretto. Si deve, invece, dichiarare dog4 come statica:

```
static char dog 4 [] = "wow";
```

Alternativamente si può utilizzare un puntatore ad un array statico (benché questo richiederà più spazio), con

```
char *dog5 = "wow";.
```

che crea un array **statico** di caratteri ed un puntatore **auto** all'array **dog5.**

Costanti in virgola mobile

La forma delle costanti in virgola mobile è:

«-»«parte intera»«parte frazionaria»«E o e»«segno»«esponente»

almeno la parte intera e la parte esponenziale o la parte frazionaria devono essere presenti. Lo spazio non è consentito all'interno. Le costanti in virgola mobile comprendono quelle di tipo **double.** Esempi corretti di costanti in virgola mobile sono:

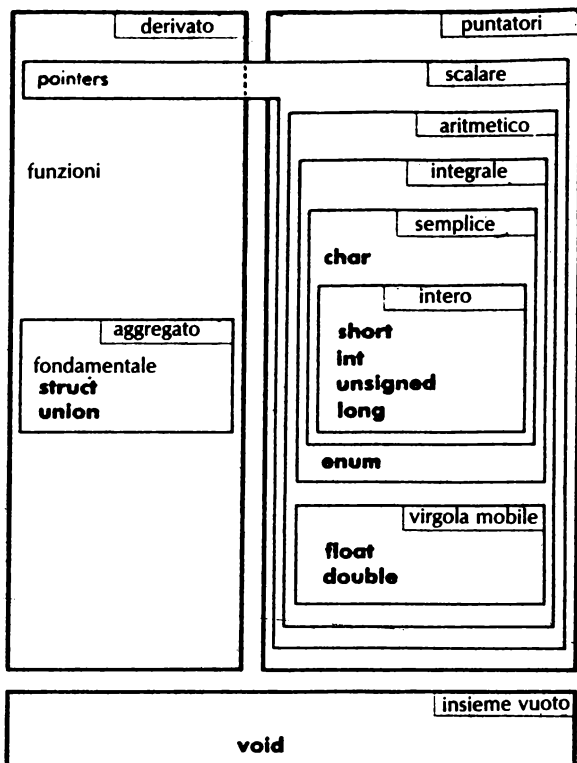
0.7e5
19e - 56
19E + 56
351.2106

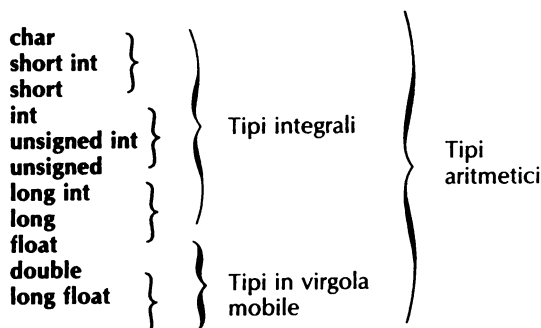
Nota che '+351.2106' non è una costante in virgola mobile corretta dato che il segno più in C non è utilizzato con questo scopo.

Variabili e dichiarazioni

Tipi di dati

I tipi di dati nel C e la terminologia usata per descriverli sono visualizzati in fig. 1

Fig. 1 Tipi di dati

Tipi aritmetici

I tipi di dati base del C linguage sono:

△ Alcuni calcolatori permettono l'uso di altre combinazioni di short o long o privi di segno (per esempio unsigned char). La maggior parte di queste combinazioni non sono uguali per tutti i calcolatori. Per utilizzarle si deve usare la **typedef.** La lunghezza di ogni tipo di dato base e la memoria associatagli, dipende dal calcolatore. Le lunghezze minime sono illustrate a pag. 44.

Dichiarazioni

Le dichiarazioni specificano la classe di appartenenza, il tipo, il nome, e le inizializzazioni opzionali di un elemento. La tabella 1 indica la sintassi di una dichiarazione semplice.

Tabella 1 semplici dichiarazioni

<sp cl>	<spec. tipo>	<dichiarato-re>	<inizializzato-re>
	char		
	short		
	int		
auto	***long***	*id*	
static	***unsigned***	**id*	
register	***float***	*id()*	*inizializzatore;*
extern	***double***	*id[constant expr]*	
	struct-spec		
	union-spec		
	enumu-spec		
	typedef-name		

Specificatori classe di appartenenza

Una dichiarazione all'esterno di una funzione sottintende la classe **static.** Una variabile statica ha il posto assegnato per tutta la durata del programma e il suo valore rimane lo stesso a meno che venga cambiato da una funzione: il suo valore è accessibile per tutto il programma. Le parole chiave **auto** sono sovrabbondanti in quanto le variabili **auto** non possono essere dichiarate all'esterno di una funzione e le dichiarazioni all'interno di una funzione sono considerate **auto** in ogni modo. La locazione di memoria è assegnata sino al ritorno della funzione: il valore di una **auto** è accessibile all'interno della funzione.

Su alcuni calcolatori, elementi **static** sono elaborati più velocemente e efficacemente che gli elementi **auto.** Anche le parole chiave **register** creano elementi **auto** ma esse ordinano al compilatore di memorizzare un elemento in un registro veloce; è un avviso per il compilatore che una variabile è usata frequentemente. I compilatori attuali permettono a scalari (ma non tipi in virgola mobile) di essere memorizzati in registri. Se nessun registro è disponibile, la parola chiave è ignorata e la variabile è convertita in **auto.**

L'indirizzo di una variabile **register** è sconosciuto.

Le variabili **extern** non occupano posto in memoria ma indicano che l'oggetto dichiarato è presente o più avanti nel file considerato o in un altro file sorgente, una più completa discussione sulla visibilità è a pag. 39.

Specificatori di tipo

Gli specificatori di tipo indicano il tipo a cui l'elemento considerato appartiene. Se lo specificatore di tipo è omesso è sottointeso il tipo **int.**

Dichiaratori

Un identificatore di per sè dichiara qual è il nome di un elemento specificato. Cosi **static int x,** dichiara un intero statico chiamato x. Inoltre asterischi, parentesi rotonde e parentesi quadre, possono essere usate per dichiarare rispettivamente puntatori a un elemento, funzioni ritornanti un elemento e array di elementi (vedi pag. 17 e seguenti).

Inizializzatori

Può essere presente un inizializzatore singolo per ogni elemento dichiarato. Per esempio:

int y = 34;

Espressioni variabili possono essere usate per inizializzazioni se la dichiarazione ricorre all'interno di una funzione come in

int z = x+y;

Per array e strutture (le unioni non possono essere inizializzate), l'inizializzatore è composto da un insieme di inizializzatori,

separati da virgola, per gli elementi aggregati scritti nell'ordine crescente degli elementi. Questa regola potrebbe essere applicata ricorsivamente per sottoaggregati. Se ci sono meno inizializzatori che elementi a quelli sovrabbondanti viene assegnato lo zero. Un semplice caso è:

static double x[3]= { 1.0,2.0,3.0 };

Le parentesi graffe possono essere omesse all'interno ma nota che mentre

static double y[3][2] = {{1.0},{1.0},{1.0}};

inizializza **y[0][0],y[1][0]** e **y[2][0]**
la dichiarazione

static double z[3][2] = {1.0,1.0,1.0};

inizializza **y[0][0], y[0][1]** e **y[1][0]**. Array di caratteri possono essere inizializzati con delle stringhe.

Uso di enumerazioni, strutture e union tag.

La sintassi per le enumerazioni, strutture e unioni è identica. Esistono due forme che possono essere usate. Primo, il tipo di composizione può essere completamente dichiarato in situ, con o senza l'uso di un tag.

enum <tag> {dichiarazione} è un *enum-spec*
struct <tag> {dichiarazione} è un *struct-spec*
union <tag> {dichiarazione} è un *union-spec*

Se il tag è dato può essere successivamente utilizzato per realizzare la definizione completa, cosi

enum tag diventa un *enum-spec*
struct tag diventa una *struct-spec*
union tag diventa una *union-spec*

Enumerazioni

△ Le enumerazioni non sono utilizzabili in molti compilatori. Esse definiscono un set di valori che saranno visti come un tipo distinto dai tipi integrali, benché un'enumerazione variabile goda della maggior parte delle proprietà di un **int.**

La sintassi di uno specificatore di enumerazione è
enum <tag> *<lista enum>*
con la lista avente sintassi: *identificatore <= espressione costante>,...*

Ogni identificatore dà un nome a un valore del set dell'enumerazione. Il primo valore è imposto a 0; ogni successivo valore assume il valore del suo precedente più uno. Questi assegnamenti sono ignorati se è presente un'espressione costante. Così:

enum {red,green,blue} x = red;

alloca della memoria per una variabile enumerata **x** la quale può assumere i valori red, green o blue (0, 1 o 2).

enum {red=-1, green, blue} y=red;

crea una variabile simile che assume i valori -1, 0 e 1. È possibile che un'enumerazione contenga valori costanti doppi ma non può contenere identificatori duplicati. La visibilità delle enumerazioni di identificatori è la stessa dei nomi di variabili ordinarie e può essere in contrasto con esse. Così

int green;

può essere in contrasto con gli esempi qui sopra. Solo l'operatore di assegnamento **=** e gli operatori logici **==** e **!=** sono permessi per le enumerazioni di variabili.

Strutture

Una struttura è una sequenza di valori variabili i quali possono essere di tipi diversi. La sintassi di una specificazione di struttura è:

struct <tag> *<member declaration list>*.

Dove la member declaration list è una lista di una o più variabili o dichiarazioni di bitfield. Strutture di tipo **auto** non possono essere inizializzate.

In alcuni casi, le strutture possono essere assegnate e possono essere inviate o ricevute da funzioni.

Componenti di struttura — variabili

Le componenti di strutture possono essere variabili di ogni tipo, ma una struttura non può contenere un'altra struttura simile a sè. Cioè se **bug** è il tag di una struttura:

struct bug {int x; struct bug y;}

è un errore; si possono utilizzare, però, puntatori alla stessa struttura:

struct nobug {int x; struct nobug *y;}

Le componenti di struttura sono riferite al contenuto di una struttura, cosi

nobug.x

si riferisce all'elemento **x** di una struttura (in questo caso a un **int**). Se **ps** è un puntatore a **nobug**, allora **(*ps).x** si riferisce allo stesso oggetto. Le parentesi sono necessarie perché l'operatore punto (member) ha precedenza nei confronti dell'operatore asterisco (puntatore a). Una alternativa equivalente ma più conveniente è:

ps->x

dove il segno -> è formato da una combinazione del segno meno e della parentesi angolata.

Componenti di struttura - bitfield

Per poter essere utilizzate su tutti i calcolatori le dichiarazioni di bitfield devono assumere la forma

unsigned *<identificatore>: ampiezza campo*

dove ampiezza campo è una costante positiva non più grande del numero di bit di un **unsigned int.** I bitfield non sono memorizzati all'interno dei limiti di un **int;** puntatori a bitfield, array di bitfield e funzioni ritornanti bitfield non sono permessi. I bitfield vengono utilizzati per risparmiare la memoria. Per esempio,

struct {int red, int blue, int green, int bright;}

richiede l'uso di quattro **int** in memoria, mentre

struct {unsigned red: 1, blue: 1, green: 1, bright: 1;}

richiede l'uso di un solo **int.** I bitfield sono anche utilizzati come componenti di struttura e non possono essere inizializzati.

Unioni

Le unioni definiscono una variabile la quale contiene esattamente una variabile di un set di variabili diverse tra loro. Le unioni hanno la stessa forma delle strutture ma cominciano con la parola chiave **union.** I bitfield non sono permessi all'interno di unioni. Per esempio:

Union struct {int i, float f;} both;

dichiara un unione che memorizza o un **int** o un **float.** È conve-

niente conoscere sempre ciò che è memorizzato in una unione; così

```
int x:
...
both.f=12.34
x=both.i;
```

assegnerà un valore indefinito alla **x**. Le unioni non possono essere inizializzate, anche se esse appartengono alla classe **static**

Puntatori

Un asterisco precedente un identificatore in una dichiarazione, dichiara un elemento del tipo 'puntatore a', così **char c;** dichiara un elemento di tipo **char** mentre, **char** *c; dichiara un puntatore a un elemento di tipo **char.** I puntatori possono puntare a qualsiasi variabile ma non a **bitfield** o a variabili **register.** Un subset di operazioni aritmetiche possono essere eseguite sui puntatori. Interi possono essere sommati o sottratti dai puntatori (incrementi e decrementi sono casi speciali). Il significato dell'espressione

```
char *P;
P=P+1;
```

è «punta al prossimo elemento di tipo char» e affinché questo abbia significato, si deve garantire che il prossimo elemento in memoria sia anch'esso di tipo **char.** Questo è il solo caso in cui un puntatore punti ad un array. Un puntatore può essere assegnato a un altro puntatore, o ad una **int** purché essa sia abbastanza grande (il tipo **long** lo è sempre, il tipo **int** potrebbe non esserlo). I puntatori possono essere comparati a altri puntatori o a NULL; il puntatore NULL non punta a nessun elemento valido. I puntatori a elementi dello stesso array possono essere sottratti. Le differenze tra i tipi sono sempre dipendenti dal calcolatore. Non è necessariamente vero che un puntatore ad un elemento di un

tipo abbia lo stesso formato di un puntatore ad un elemento di un altro tipo. Puntatori NULL che sono assegnati a una funzione dovrebbero essere forzati al tipo corretto. Così **execl ("myprog", "- bug", NULL)** non funzionerebbe su alcuni sistemi perché passa l'intero NULL a **execl,** ciò che è richiesto è specificare esplicitamente il tipo come in **execl ("myprog", "- bug", (char *)0)**

Questo è il modo preferibile per dichiarare un puntatore nullo, perché (per esempio) un puntatore a una funzione può avere una dimensione che è differente dalla dimensione di un puntatore a un **char.**

Array

Un identificatore che precede parentesi quadrate e una espressione costante dichiara un elemento del tipo «array di...» così

char x [100];

dichiara un array di 100 **char.** Gli array che appartengono alla classe **auto** non possono essere inizializzati. Come un caso speciale di creazioni ricorsive di nuovi tipi di dati, **char x [25][80];** crea un array di dimensione da 25 per 80. Gli array sono memorizzati per riga, così

char x [0] [0...79]
char x [1] [0...79]

sono sicuramente contigui.

Dichiarazioni di funzioni

Un identificatore che precede parentesi rotonde vuote dichiara un oggetto del tipo "funzione che ritorna...", così,

float x ();

dichiara una funzione **x** che ritorna un elemento di tipo **float.** △

Le funzioni possono essere dichiarate con variabili di tipo **void,** il che significa che la funzione non dà in uscita un valore e che è scorretto utilizzare un valore come risultato della funzione. Le funzioni possono avere valori di tipo aritmetico, enumerazioni, puntatori e in alcuni sistemi △ strutture.

Esse possono appartenere solo alle classi **extern** o **static.** Se il compilatore incontra un identificatore precedentemente indefinito, seguito da una parentesi a sinistra, esso lo dichiarerà contestualmente come una funzione determinante **un int:** per cui riferimenti a tali funzioni precedenti alla loro dichiarazione sono permessi. Riferimenti a una funzione che non determina un **int** devono essere dichiarati esplicitamente. È praticamente impossibile scrivere una funzione indipendente dal calcolatore su cui è applicata, con un numero variabile di argomenti. L'ordine di valutazione degli argomenti di una funzione è indefinito, cioé

```
func (x, y) int x,y; {...}
int a=1;
func (++a, ++a)
```

non è attendibile e potrebbe risultare x=2 e y=3 o viceversa. Gli argomenti delle funzioni sono sempre forzati in uno dei **tipi int, long, double,** *puntatori a o* △ strutture, prima della chiamata della funzione. Non ha importanza, quindi, se si chiama una funzione con un **float** o una **double,** o (soggetto a una possibile estensione del segno) un **char** o un **int.** △ l'ANSI standard Committee (x3j11) per il C ha definito la possibilità di effettuare dei controlli sul tipo dei argomenti delle funzioni (attualmente esistente solo nel MC). Questo significa che il compilatore può essere avvisato quando l'argomento di una funzione dovrebbe essere diverso dai parametri attuali e che può scoprire dei conflitti nel numero dei parametri attuali e dei parametri formali; il compilatore può anche forzare i parametri attuali quando vi è un conflitto di tipo. La sintassi definita per il controllo dei tipi dei parametri è uguale a quella delle dichiarazioni di tipo; per esempio:

△ **int check (int, double);**

la quale dichiara che **check** prende due parametri e che essi sono di tipo **int** e **double** rispettivamente. Il caso speciale di una funzione senza parametri è dichiarata come

△ **int nopart (void);**

che è diverso da

int nocheck ();

che è la vecchia sintassi e disabilita ogni controllo che il compilatore potrebbe altrimenti fare. Le funzioni che utilizzano un numero variabile di argomenti possono essere dichiarate come

△ **int varang (char*,);**

Nell'esempio dato **varang** contiene un parametro di tipo puntatore a **char** e qualunque argomento di tipo sconosciuto (e non controllato) (la funzione di libreria **printf** potrebbe essere dichiarata in questo modo.)

Dichiarazioni complesse

In generale tutte le dichiarazioni citate sopra possono essere applicate ricorsivamente. Il seguente è un facile metodo manuale per scrivere dichiarazioni complesse.

1) Scrivere la dichiarazione in italiano, usando le espressioni "puntatore a", "funzione ritornante", e "array di" (per esempio: "funzione ritornante un puntatore a un arry di puntatori a struttura")

2) Scrivere il tipo alla sinistra, il nome nel mezzo e il punto e virgola alla destra, p.e.: **struct sample** x

3) sostituisci i termini
***(termine)** invece di "puntatore di"
(termine) () invece di "funzione ritornante"
(termine) [] invece di "array di"

Nel nostro esempio (una funzione che determina un puntatore a un array di puntatori a struttura) si ottiene

```
struct sample        *(x)            ;
struct sample        (*(x))[]        ;
struct sample        *((*(x))[])     ;
struct sample        (*((*(x))[]))() ;
```

Un metodo simile può essere usato per ridurre dichiarazioni complesse:

1) imprimersi nella mente che gli operatori primari [] e () hanno precedenza sul segno * a partire da sinistra verso destra. (quindi rimuovere le [] o () più a destra se vi è una combinazione di questi operatori). Le parentesi possono cambiare l'ordine di un istruzione.
2) scrivere "funzione che ritorna" invece di ()
   "array di" invece di []
   "puntatore a" invece di *
   e cancellare gli elementi corrispondenti delle dichiarazioni.
3) ripetere il secondo passo finché solo l'identificatore è a sinistra e si ha la descrizione dell'identificatore.

La tabella 2 rappresenta le più comuni dichiarazioni complesse fino al quinto livello di inserimento. Essa può essere usata semplicemente per sostituzioni; prendendo la prima per esempio

```
struct test *x[];
```

dichiara che x è un array di puntatori a una struttura chiamata **test.**

Tabella 2

Secondo livello	
*x[]	array di puntatori a
*x()	funzioni ritornanti puntatori a
(*x)[]	puntatori ad arry di
(*x)()	puntatori a funzioni ritornanti
Terzo livello	
(*x[])[]	array di puntatori a array di
(*x[])()	array di puntatori a funzioni ritornanti
(*x())[]	funzioni ritornanti puntatori a array di
(*x())()	funzione ritornante puntatore a funzione ritornante
*(*x)[ ]	puntatori a array di puntatori a
*(*x)()	puntatori a funzioni ritornanti puntatori a
Quarto livello	
*(*x[])[]	array di puntatori a array di puntatori a
*(*x[])()	array di puntatori a funzioni ritornanti puntatori a
*(*x())[]	funzione ritornante puntatore a array di puntatori a
*(*x())()	funzione ritornante puntatore a funzione ritornante puntatore a
(*(*x)[])[]	puntatori a array di puntatori a array di
(*(*x)[])()	puntatori a array di puntatori a funzioni ritornanti
(*(*x)())[]	puntatori a funzioni ritornanti puntatori a array di
(*(*x)())()	puntatori a funzioni ritornanti puntatori a funzioni ritornanti
Quinto livello	
(*(*x[])[])[]	array di puntatori a array di puntatori a array di
(*(*x[])[])()	array di puntatori a array di puntatori a funzioni ritornanti
(*(*x[])())[]	array di puntatori a funzioni ritornanti puntatori a array di
(*(*x[])())()	array di puntatori a funzioni ritornanti puntatori a funzione ritornante
(*(*x())[])[]	funzione ritornante puntatore a array di puntatori a array di
(*(*x())[])()	funzione ritornante puntatori a array di puntatori a

	funzioni ritornanti
(*(*x())())[]	funzione ritornante puntatori a funzione ritornante puntatore a array di
(*(*x())())()	funzione ritornante puntatore a funzione ritornante puntatore a funzione ritornante
((*x)[])[]	puntatori a array di puntatori a array di puntatori a
((*x)[])()	puntatori a array di puntatori a funzioni ritornanti puntatori a
((*x)())[]	puntatori a funzioni ritornanti puntatori a array di puntatori a
((*(x)())()	puntatori a funzioni ritornanti puntatori a funzioni ritornanti puntatori a

Typedef

I typedef possono essere usati per creare nuovi nomi di tipi di dati. Ciò è utile per parametrizzare i programmi contro i problemi di portabilità, e per semplificare dichiarazioni non intuitive come

float * (*x())();

Sintatticamente **typedef** appare al posto del tipo predefinito e il nome del nuovo tipo di dati al posto dell'oggetto dichiarato. Per cui

typedef float * (*MYTYPE())();

permetterà di utilizzare **MYTYPE** per dichiarare funzioni determinanti puntatori a funzioni ritornanti puntatori a **float** e inoltre

extern MYTYPE a,b,c;
diventa

extern float * (*a())(), * (*b())(), *(*c()();

Alcune typedef potrebbero essere combinate come in

typedef char *STRING, *SPUNC(), *(*SFARR[])();

il quale definisce tre nuovi tipi di stringhe, stringhe (puntatori a **char**), funzioni determinanti stringhe e array di puntatori a funzioni determinanti stringhe.

Espressioni e operatori

La figura 2 indica gli operatori del linguaggio C e le loro precedenze. Nessuno spazio è consentito tra i caratteri di operatori multicarattere.

Fig. 2 Classi di oper. e precedenze

<table>
<tr><th>Categoria</th><th>Operatore</th><th>Associatività</th></tr>
<tr><td>primario</td><td>( ) [ ] . -></td><td>da sin. a des.</td></tr>
<tr><td>unary</td><td>* & ++ --
~ - sizeof (nome-tipo)
!</td><td>da des. a sin.</td></tr>
<tr><td rowspan="11">binario</td><td>* / %</td><td rowspan="11">da sin. a des.</td></tr>
<tr><td>+ -</td></tr>
<tr><td><< >></td></tr>
<tr><td>< > <= >=</td></tr>
<tr><td>== !=</td></tr>
<tr><td>&</td></tr>
<tr><td>^</td></tr>
<tr><td>|</td></tr>
<tr><td>&&</td></tr>
<tr><td>| |</td></tr>
<tr><td>condizionale</td><td>?:</td><td>da des. a sin.</td></tr>
<tr><td>assegnamento</td><td>= += -= *=
/= %= >>= <<=
&= ^= | =</td><td></td></tr>
<tr><td>virgola</td><td>,</td><td>da des. a sin.</td></tr>
</table>

Operatori primari

Gli operatori primari sono già stati trattati a pag. 18 (funzioni e arry), e 15 (strutture e unioni).

Operatori logici

Il linguaggio C considera il valore numerico 0 come falso, qualsiasi altro valore è considerato vero. Gli operatori logici determinano sempre un valore di tipo **int:** 1 per vero e 0 per falso. Il loro significato è dato nella tabella 3; le parentesi graffe indicano un uguale livello di precedenza. (Nota che il termine 'operatori logici' è spesso usato in un senso più restrittivo per riferirsi solo agli operatori **&&** e ||)

Tabella 3

Operatore	Esempio	Risultato
!	!a	1 se a è 0, altrimenti 0
<	a<b	1 se a<b, altrimenti 0
<=	a<=b	1 se a<=b, altrimenti 0
>	a>b	1 se a>b, altrimenti 0
>=	a>=b	1 se a>=b, altrimenti 0
==	a==b	1 se a è uguale b, altrimenti 0
!=	a!=b	1 se a è non uguale a b, altrimenti 0
&&	a&&b	1 se a e b sono veri, altrimenti 0
||	a||b	1 se a è vero, (b non è valutato), altrimenti 1 se b è vero, altrimenti 0

Altri operatori

La tabella 4 rappresenta i dettagli di altri operatori che non sono primari o logici; le parentesi graffe indicano livelli di precedenza uguali. lv nell'esempio indica che l'espressione deve essere un lvalue. La lettera i nella penultima colonna indica che l'operatore esige valori interi come operandi.

Operatore di cast

L'operatore di cast *(tipo)* forza la conversione dei suoi operandi a un tipo di dati specificato. In casi semplici, il *tipo* è proprio la parla chiave per un tipo di dati, come in

(long) 5

che è lo stesso di **5l** o **5L**

In casi più complessi, il tipo è la dichiarazione di quel tipo con l'identificatore omesso (cioè un dichiaratore astratto). Così se

int (*pai)[];

dichiara che **pai** sarà un puntatore a un array di **int,** allora

(int(*)[])0

è un puntatore NULL a un array di **int.**

L'operatore Sizeof

L'espressione **sizeof (tipo)** dà come risultato una costante intera priva di segno rappresentante la dimensione in byte di un elemento del tipo in questione. La sintassi di tipo è uguale a quella data per i cast. Quando utilizzato con un array l'operatore **sizeof** indica la dimensione (in byte) dell'array.

Tabella 4 altri operatori

Operatore	Esempio	Risultato	vedi pag.
~	~a	il complemento a uno a di	
++	++lv	lv dopo incremento	
	lv++	lv prima dell'incremento	
--	--lv	lv dopo decremento	
	lv--	lv prima dell'incremento	
-	-a	negativo di a	
()	(tipo) a	a convertito al tipo	26
*	*p	elemento puntato da p	
&	&lv	indirizzo dell'elemento lv	
sizeof	**sizeof** e	ampiezza (in byte) di e	
	sizeof(t)	ampiezza in byte di t	
*	a*b	a moltiplicato b	
/	a/b	a diviso per b	
%	a%b	a modulo b	i
+	a+b	a più b	
-	a-b	a meno b	
<<	a<<b	a shiftato a sinistra b bit	i
>>	a>>b	a shiftato a destra b bit	i
&	a&b	operatore AND tra a e b	i
^	a^b	operatore XOR tra a e b	i
|	a | b	operatore OR tra a e b	i
?:	a?e1:e2	e1 se a è vero altrimenti e 2	29
=	lv=b	lv, con b assegnatogli	
+=	lv+=b	lv, lv=lv+b	
-=	lv-=b	lv, lv=lv-b	
*=	lv*=b	lv, lv=1v*b	
/=	lv/=b	lv, lv=1v/b	
%=	lv%=b	lv, lv=1v%b	i
>>=	lv>>=b	lv, lv=lv>>b	i
<<=	lv<<=b	lv, lv=lv<<b	i

&=	lv&=b	lv, lv=lv&b	i	
^=	lv^=b	lv, lv=lv^b	i	
|=	lv |=b	lv, lv=lv |b	i	
	e1, e2	e2(e1 valutato primo)		29

L'operatore condizionale

L'operatore condizionale ternario ?: controlla il suo primo operando. Se il primo operando è vero (non 0), il secondo operando viene esaminato altrimenti viene considerato il terzo operando. Così l'espressione
e1 ? e2 : e3 diventa
e2 se e1 è vero oppure e3 se e1 è falso. Le usuali conversioni aritmetiche sono eseguite su e2 e e3.

L'operatore virgola

L'operatore binario virgola considera il primo operando e scarta il risultato. Successivamente viene considerato il secondo operando che viene considerato come risultato. Così

b++, c

incrementa **b** e assume il valore **c.**

Il tipo del risultato è lo stesso del secondo operando. Se l'operatore virgola appare in una lista separata da virgola, deve essere posto tra parentesi; così

func ((b++,c));

è una chiamata a **func** con il solo argomento **c.**

L'usuale conversione aritmetica

L'usuale conversione aritmetica può essere espressa in un linguaggio un po' simile al C come segue:

convertire ogni operando **float** in **double;**
IF (un operando è di tipo **double)**

convertire gli altri operandi in **double;**
ELSE
    convertire ogni **char** in **int;**
    convertire ogni **short** in **int;**
    IF (un operando è di tipo **long)**
        convertire gli altri operandi a **long;**
    ELSE
        IF (un operando è **unsigned);**
            convertire gli altri operandi in **unsigned)**
        ELSE
            entrambi gli altri operandi sono **int.**

La regola è più complicata per i compilatori che permettono altre quantità **unsigned.** Inoltre, dal momento che un puntatore potrebbe essere equivalente a un **int** o a un **long,** il codice che utilizza una conoscenza precisa della dimensione di un puntatore potrebbe portare a risultati errati su una macchina ma non su altre.

Conversione e estensione del segno

Una quantità di tipo **int** è convertita in una più grande quantità integrale con estensione in segno, ma se è di tipo **unsigned** si utilizza lo zero-padding. △ Un risultato inaspettato può capitare se **long** e **unsigned int** hanno la stessa dimensione.

△ I caratteri possono essere utilizzati ovunque è utilizzato un **int,** e possono avere o non avere l'estensione di segno.

Questo è un inconveniente serio, dal quale è bene guardarsi attentamente. Quantità integrali più lunghe sono convertite in quantità integrali più corte per mezzo di un troncamento a sinistra. Tutti i numeri in virgola mobile sono posti in doppia precisione. L'arrotondamento si ha quando un **double** è convertito in **float.** Non si perde in precisione quando si converte **float** in **double.** La conversione di valori interi in valori in virgola mobile è funzionante correttamente ma può esserci perdita di precisione. Non è da attendersi alcun particolare trattamento della parte decimale nella conversione di valori in virgola mobile in valori

interi. Il risultato è indefinibile se la destinazione non ha bit a sufficienza.

Le funzioni di libreria **ceil, floor, frexp, ldexp, modf** alleviano alcuni di questi problemi di conversione.

Ordine di valutazione

La precedenza e l'associatività di operatori è data in figura 2 a pag. 24. Altrimenti l'ordine di valutazione è indefinito. Il compilatore è libero di calcolare sottoespressioni in qualsiasi ordine senza controllare gli effetti collaterali. Così

a[i++] = b [i++];

può o non può ottenere l'effetto desiderato. Le espressioni contenenti gli operatori commutativi e associativi *****, **+**, **|** e **^**, possono essere riordinati, così nell'espressione:

fa(x)+fb(x)

il compilatore è libero di valutare **fb** prima di **fa** anche se il risultato non sarebbe più lo stesso. Più generalmente il risultato di qualsiasi espressione che riutilizza una variabile cambiata per effetti secondari (side-effects), dovrebbe essere considerata indefinita.

Argomenti di una funzione

Gli argomenti di una funzione di tipo **float** sono convertiti in **double** prima della chiamata. Argomenti di tipo **char** o **short** sono convertiti in **int.** Nomi di array sono convertiti in puntatori al primo elemento dell'array. △ Alcuni compilatori applicano la stessa regola ai nomi di strutture.

Immaginiamo che ci sia una struttura dichiarata come

struct mixup {int a; int b;};

successivamente
fcall (& mixup)
chiamerà **fcall** con un puntatore alla struttura, ma alcuni compilatori (di quelli che non riconoscono le strutture come argomenti

di una funzione) convertiranno **fcall (mixup)** △ nella prima forma, sebbene ciò che probabilmente è inteso è che **fcall** dovrebbe essere chiamato con una copia di **mixup.**

ISTRUZIONI

Istruzioni e espressioni

Si può utilizzare qualsiasi espressione come un istruzione terminandola con un punto e virgola. Per esempio **a=3** è un espressione mentre **a=3**; è un istruzione. Ciò comporta l'effetto indesiderato che, ovviamente, istruzioni assurde sono ammesse:

```
a+3;
```

è ammessa, ma tutto ciò che fa è sommare a e 3 per poi perdere il risultato. Alcuni compilatori, e la utility lint dello UNIX segnaleranno l'errore, altri no. Ugualmente problematico è

```
*P++;
```

istruzione che incrementa il puntatore **P**, ritornando il valore puntato da esso per poi perderlo

Blocchi

I blocchi permettono a più di un istruzione di apparire dove dovrebbe esserci una sola istruzione. Un blocco di istruzioni ha la forma:

{ <lista di dichiarazione><lista istruzioni> {

La lista di dichiarazioni è una lista di dichiarazioni ristrette al blocco. Dichiarazioni precedenti di variabili con lo stesso nome sono sospese all'interno del blocco, cosi

```
int d=9;
main ()
{
```

```
	int d= 12;
	{
			int d= 15;
			/*d vale 15*/
	{
	/*d vale 12*/
{
/*d vale 9*/
```

è ammessa. Nessuna dichiarazione sarà ammessa dopo la prima istruzione.

IF e ELSE

Un istruzione condizionata ha la forma
if (espressione) istruzione <**else** istruzione>.

Se l'espressione non è 0 la prima istruzione è eseguita, altrimenti (se la clausola **else** è presente) è eseguita la seconda istruzione. Per esempio:

```
if (a==3)
	printf ("la variabile a vale 3/n");
else
	printf ("a non vale 3/n");
```

In una serie di clausole **if-else, else** è collegato all'**if** più vicino.

While

L'istruzione **while** assume la forma
while *(espressione) istruzione*

L'espressione viene controllata prima di ogni esecuzione, e l'istruzione è eseguita zero o più volte finché l'espressione non è diversa da 0. Cosi

```
int c=10;
while (c>=0)
{
```

```
        printf ("%d\n",c);
        c——;
{
```

conta decrementando da 10 a 0. Un **break** nell'istruzione passerà il controllo all'istruzione seguente. Un **continue** passerà il controllo alla prima espressione che verrà individuata.

Do... While

L'istruzione **do** assume la forma
do *istruzione* **while** (*espressione*);

L'espressione è valutata dopo ogni esecuzione e l'istruzione è eseguita una o più volte finché l'espressione è 0. Per esempio per controllare gli spazi dallo standard input si può scrivere

```
int c, nsp;
do
{
 c=getchar();
 if (c==' ') nsp ++;
}while (c!=EOF);
printf ("%d spazi\n",nsp);
```

Un **break** nell'istruzione passerà il controllo all'istruzione successiva. Un **continue** passerà il controllo alla valutazione della prossima *espressione.*

For

L'istruzione **for** assume la forma
for (<e1>;<e2>;<e3>;) **istruzione**
dove e1, e2, e3, sono delle espressioni.

L'istruzione **for** è equivalente a

```
e1;
while(e2)
{
        istruzione
CONT:e3;
{
```

con la clausola che un **continue** nell'istruzione trasferisce il controllo alla label CONT. Ognuna delle espressioni può essere omessa. Se si omette e 2 il test è sempre vero, per cui l'istruzione
for (e1;;e3)
è un ciclo infinito che può terminare solo con **break, return** o **goto.** Un **break** passerà il controllo all'istruzione successiva. Un **continue** passerà il controllo alla prossima esecuzione di e3. Un tipico uso del **for** è

```
# define NEL 100
float x[NEL];
int i;
for (i=0; i<NEL; i++);
x [i]=99.0;
```

che inizializza un intero array **x** con il valore in virgola mobile 99.0.

Switch

L'istruzione **switch** ha la forma
switch (espressione) **switch-block**
dove switch block è un blocco in cui la lista di dichiarazione non contiene variabili **auto** o **register.** All'interno di uno switch-block ci possono essere case label la cui sintassi è

case espressione costante:

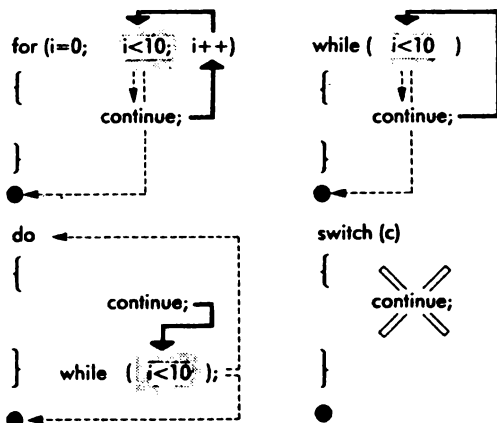
L'espressione intera **switch** è valutata e se si accoppia a qualsiasi delle espressioni-costanti in un case-label, l'istruzione seguente la label viene eseguita. Se non c'è un riconoscimento di espressioni-costanti, il controllo passa alle istruzioni seguenti lo switch-block, a meno che il block contenga la label **default:** In questo caso il controllo passa all'istruzione seguente la **default.** Se ci sono due case-label nello stesso blocco non possono avere espressioni costanti con lo stesso valore. Se il controllo raggiunge qualsiasi istruzione nel ciclo switch esso procede sequenzialmente da quell'istruzione in avanti. L'istruzione **break** può essere usata per passare il controllo all'istruzione seguente lo switch-

block. L'istruzione **continue** non si riferisce mai a una **switch** ma può essere usata per controllare esecuzioni di cicli **do, while** o **for** che lo contengono.

L'esempio seguente illustra un tipico uso di **switch** (senza nessuna prova per controllare gli errori ritornati da **gets**):

```
char answer [100];
char *gets ();
printf ("Prego rispondere Si o No");
gets (answer);
switch (answer[0])
{
    case 'S':
    case 's': printf ("\nHai risposto si\ n");
              break;
    case 'N':
    case 'n': printf ("\nHai risposto no\n");
              break;
    default: printf ("\nPrego prova ancora\n");
              break;
}
```

Break e continue — sommario

Si usa l'istruzione **break** nell'istruzione **switch** e nei cicli **do, while** e **for.** Si può usare l'istruzione **continue** solo per completare i cicli **do, while** e **for.** Il diagramma del controllo è rappresentato in fig. 3, il pallino nero indica la destinazione di un **break**

Return

L'istruzione **return** assume la forma:

return <*espressione*>

Essa causa il ritorno immediato da una funzione. Se l'espressione è presente essa è convertita al tipo con cui la funzione è stata dichiarata. Se non è presente alcuna espressione, il valore ottenuto è indefinito. Lo stesso accade se la funzione termina senza un **return.** Per esempio,

```
cvt (d)
int (d);
{
        if ( (d>='0') && (d<='9') )
        return (d—'0');
{
```

dà un risultato indefinito a meno che **d** sia una cifra.

Goto

L'istruzione **goto** assume la forma

goto *identificatore*

dove **identificatore** è un riferimento a una label. Una label assume la forma:

identificatore :

e può precedere qualsiasi istruzione. Il suo scopo è la funzione corrente; non è possibile saltare da una funzione all'altra (vedi, però, **setjmp** e **longjmp** nella sezione libreria). È consentito, invece, saltare in un block interno nella stessa funzione, ma se la label non precede la declaration-list di quel blocco qualsiasi inizializzazione di variabili **auto** o **register** non verrà eseguita. Così la seguente codifica non è perfettamente sicura:

```
func ()
{
        goto noval
{
        int x=12;
        noval :;/*x non è definito */
        }
}
```

L'istruzione vuota

L'istruzione vuota

;

è consentita ed è spesso usata dopo un ciclo **for** quando tutte le operazioni del ciclo sono state eseguite all'interno del ciclo stesso.

Identificatori

Visibilità

È importante comprendere la differenza tra lo scopo di una variabile e la visibilità, o lo scopo lessicale di un identificatore. Una variabile è **static** o **local** e questo indica la durata della sua vita. Un identificatore è detto visibile se il suo tipo di appartenenza e il suo nome sono conosciuti in un punto particolare del file sorgente. Le label sono visibili in tutta la funzione in cui esse sono dichiarate, anche se esse sono poste in un blocco interno. Sono permessi riferimenti a label definite in seguito.

Le regole per variabili, funzioni, costanti **enum** e nomi di **typedef** sono:

- La visibilità di un identificatore definito fuori da ogni funzione e la visibilità delle funzioni stesse sono il resto del file sorgente. Per lo scopo della visibilità, file **# include** sono parte del file sorgente.
- La visibilità di un identificatore definito in un blocco è il resto di quel blocco, includendo i sottoblocchi.
- La visibilità di un parametro formale di una funzione è l'intero corpo della funzione.
- Riferimenti a identificatori da definire non sono permessi. In pratica, c'è un'eccezione, nel senso che è consentito il riferimento ad una funzione definita in seguito, che abbia un **int** come valore: quando un compilatore incontra un identificatore sconosciuto seguito da una parentesi a sinistra, l'identificatore è contestualmente dichiarato **extern int id ();**
- Variabili e funzioni **static** sono visibili in altri file sorgenti nello stesso programma, se esse sono dichiarate come **extern** nei file sorgente (nota, ancora l'eccezione relativa a dichiarazioni di funzioni). Variabili esterne e funzioni che sono esplicitamente dichiarate come **static** non sono visibili all'esterno del loro file sorgente. L'apparente ambiguità della parola chiave **static** è concepibile se si considera che non ha senso avere una variabile **auto** visibile fuori di una funzione, dato che essa non esiste.

Conflitti tra nomi di variabili

Ci sono quattro classi di identificatori che non creano conflitti l'una con le altre; esse sono rappresentate in fig. 4

Tag di struttura Tag di enum, Tag di unione,	Identificatori ordinari, nomi delle funzioni costanti di enum nomi di typedef	nomi dei componenti delle strutture, nomi dei componenti di union	etichette

Fig. 4 Classi di identificatori

Il Preprocessor

Concettualmente, il C preprocessor è un filtro che è attivato prima del compilatore. Esso esegue sostituzioni di testo, inclusioni di file ed è in grado di eseguire elaborazioni condizionate. Tutte le direttive del preprocessor cominciano con il segno # che è posto all'inizio di una linea.

 # Define

Ci sono due forme della direttiva **# define.** La prima è

 # define identificatore <token-string>

e fa si che l'identificatore venga rimpiazzato dalla token string ogni volta che capita nel file sorgente. Se non è presente la token string, l'identificatore è effettivamente omesso dal file sorgente. La seconda forma della direttiva **# define** è

 # define identificatore (identificatore...) <token-string>

Non è consentito porre spazi tra il primo identificatore e la parentesi. Questa forma è capace di effettuare sostituzioni più complesse, cosi

 # define min(A,B) A<B?A:B

definisce una sostituzione per l'espressione **min(A,B)** che è il minore dei due argomenti. Siccome si sta trattando con sostituzioni testuali, gli argomenti della macro possono essere di qualsiasi tipo per i quali sono ammesse tali operazioni.

Esistono comunque due problemi. Una macro potrebbe avere anche effetti collaterali; così **min (++a,b)** viene espansa come:

```
++a<b?++a:b
```

che esegue più operazioni di quelle che dovrebbe. Secondo, siccome un identificatore potrebbe essere rimpiazzato da un'espressione contenente operatori è peferibile porre tra parentesi ogni identificatore. Così

```
# define min (A,B) ((A)<(B)?(A):(B))
```

è più sicuro e funzionerà correttamente, per esempio nel caso di

```
X+min (a+b,c+d)
```

Le # **define** possono essere poste una all'interno dell'altra.

 # Undef

La direttiva # **undef** non definisce (fa sì che il preprocessor dimentichi) un identificatore precedentemente definito. È generalmente convenuto che il preprocessor memorizza e perde dati con una tecnica LIFO (Last In First Out)

```
# define SIZE 100
# define SIZE 200
# undef SIZE
```

SIZE è definito come 100. La direttiva # **undef** cancella la definizione di identificatori dichiarati indipendentemente dal fatto che sia stata usata la prima o la seconda forma della # **define**

 # Include

La direttiva # **include** assume la forma

include *"filename"* oppure
include *<filename>*

La linea è rimpiazzata con l'intero contenuto del *filename*. Le due differenti forme possono essere trattate identicamente o meno. In entrambe i casi i file sono ottenuti da alcuni posti predeterminati nel file-system; la prima forma, probabilmente, cerca prima nella vostra directory. Come regola si usa le <> per includere file che sono rilevanti per l'intero sistema, per esempio

 # include <stdio.h>

si usano le virgolette invece per includere file specifici, come in

 # include "dogfood.h"

I file # include possono essere posti uno all'interno dell'altro.

#if, #else, #endif
La direttiva condizionale **#if** ha la forma
#if *espressione restrittiva costante*

code 1
<**# else**
code 2>
#endif

Se l'espressione costante vale vero, viene reso effettivo il codice 1 (code1), altrimenti è effettivo il codice 2 (code2). L'*espressione restrittiva costante* è un'*espressione costante* che non può contenere **sizeof**

#ifdef, #ifndef
#ifdef e **#ifndef** possono apparire al posto della direttiva **#if;** la loro sintassi è

#ifdef identificatore

se l'identificatore è (non) definito la direttiva è equivalente a **#if 1**. Alcuni compilatori permettono di usare le forme equivalenti

△ **# if defined** identificatore
△ **# if defined** identificatore

 # line
La forma della direttiva **# line** è

 # line costante filename

essa fa si che il compilatore mandi in uscita i messaggi di errore riportando il numero di linea indicato e il filename. È tipicamente usata solo dai traduttori e altri preprocessor.

Limitazioni

La tabella 5 rappresenta la misura minima delle variabili trattate lungo il testo ed inoltre alcuni limiti pratici del compilatore.

Tabella 5

Tipo	Dimensione	Intervallo
char	8 bit	0...255 o -128...127
int	16 bit	-32768...32767
short	16 bit	-32768...32767
unsigned	16 bit	0...65535
long	32 bit	-2147483648...2147483647
Tipo	Precisione	Intervallo
float	6 digit	±1.0E±36
double	13 digit	±1.0E±36

Limitazioni del compilatore	
lunghezza massima di una linea del codice:	128 caratteri
dimensione massima delle **union:**	76 membri
inserimenti di **# include:**	3 livelli
lunghezza buffer **printf:**	128 caratteri
numero dei parametri delle funzioni:	32 livelli
lunghezza delle stringhe "...":	79 caratteri
massimo programma linkabile:	300 esterni
istruzione **switch:**	32 "case"
annidamenti **while/do/for:**	10 livelli

La libreria

Il linguaggio e la libreria

Il C di per sè non definisce le funzioni di output e di input. Questi programmi sono contenuti nella libreria, che è una collezione di files **#include** e di funzioni oggetto di libreria. Esiste un ragionevole grado di standardizzazione. I maggiori problemi su un sistema che non sia UNIX sono l'apertura di file, la fine file e l'elaborazione di file non di testo.

La funzione 'main'

Un certo ammontare di elaborazioni dipendenti dai sistemi (ma trasparenti) è eseguito prima dell'esecuzione del programma

C. Per convenzione la routine di partenza esegue una funzione chiamata **main,** che potrebbe essere come la seguente:

```
main (arg,argv,envp)
int argc;
char*argv[];
char*envp[];
{
    while(--argc<0)
    procarg(*++argv);
}
```

Il primo argomento è il numero di parametri passati al programma dal processo chiamante. Il secondo argomento è un array di puntatori a stringhe che rappresentano gli argomenti passati. Utilizzando UNIX (ma generalmente non su altri sistemi) la prima di queste stringhe è il nome del programma chiamato. Le altre stringhe sono gli argomenti del programma. Per esempio, se l'utente digita **cat one two,** allora

argc	è 3	
argv[0]	punta alla stringa 'cat'	△
argv[1]	punta alla stringa 'one'	
argv[2]	punta alla stringa 'two'	

Il terzo argomento della funzione principale **envp** è utilizzabile generalmente solo sul sistema UNIX ed è un array di puntatori a stringhe d'ambiente come **PATH=:/bin :/usr/bin**

Nota che **main(), main (argc)** e **main (argc, argv)** sono ammesse

Convenzioni

L'header per ogni funzione di libreria o macro è data così:

①	②	③	④	⑤	⑥
double	fabs	(x)	math	function	math.h
double x; ⑦					⑧ △ KR CR LAT

① indica il tipo del valore ottenuto nella funzione **(double).** Un valore di ritorno di tipo **void** significa che la funzione non ritorna un valore

② fornisce il nome della funzione (fabs).

③ fornisce un esempio di argomenti per la funzione.

Alcuni argomenti comuni hanno dei nomi speciali, così che essi non necessitano di essere spiegati per ogni funzione.

④ Indica il tipo di libreria della funzione (math.h). Ciò ha poca importanza se non su sistemi UNIX. Casi possibili sono *system* (per chiamate del sistema UNIX), *math* (per le librerie matematiche), *library* per le funzioni di libreria in generale e *standard* per le funzioni che sono listate come parti dell'I/O standard di libreria in *UNIX programming* di Kernighan e Ritchie (UNIX programmer's manual, vol. 2, pp. 301-22).

⑤ indica se l'item in questione è una macro o una funzione. Item dichiarati come macro possono infatti essere implementati come funzioni ma è più prudente considerare che si sta lavorando con una macro.

Le macro hanno effetti collaterali e non potete accedere al loro indirizzo.

⑥ Indica l'header file che dovrebbe essere incluso con una # **include.**

Questo è obbligatorio, eccetto per i file math.h e string.h che dichiarano le funzioni appropriate come **extern** con l'appropriato valore di ritorno per risparmiare lavoro.

⑦ Fornisce il tipo degli argomenti. A causa delle regole di conversione per le funzioni, argomenti di tipo **float** e **double** sono intercambiabili.

⑧ Lista i compilatori per cui questa funzione non è implementata, così si può giudicare quali rischi prendere nell'utilizzare tale funzione. Tutte le funzioni che sono comuni ad almeno 6 degli 11 compilatori (considerando il libro e l'articolo di Kernighan e Ritchie come una implementazione) sono state incluse, sebbene si sono fatte alcune eccezioni. Noi abbiamo cercato di descrivere tutti i compilatori, ma occasionalmente alcune parti sono state ignorate.

Argomenti comuni

Alcuni argomenti sono comuni a un certo numero di funzioni, e, quindi, non verranno spiegati ogni volta. Essi hanno nomi che sono specifici per tutto il manuale e sono:

Fpath una stringa designante un filename o un file path ad esempio
char * Fpath="myfile"

Dpath una stringa designante un directory path ad esempio
char * Dpath="user/bin";

StreamP Un puntatore a un elemento di tipo FILE (che è dichiarato in stdio.h). Puntatori a file sono utilizzati per riferirsi a buffered stream. Nota che i nomi predefiniti **stdin, stdout** e **stderr** sono puntatori a file costanti. Puntatori a file potrebbero essere dichiarati come
FILE * Filp;

Handle I file Pointer si riferiscono a stream, un handle o descrittore di file, è un piccolo intero positivo che si riferisce a un unbuffered file aperto con una delle chiamate del sistema. Per esempio
int handle;
handle=open ("Myfile", mode);
È un grave errore usare StreamP al posto di Handle o viceversa.

Pfmt **(printf)** formato
Plst **(printf) lista di argomenti**
Sfmt **(scanf)** formato
Slst **(Scanf)** lista di argomenti (vedi pag. 48, 49)

Ritorno di errori

Molte funzioni ritornano un codice di errore il cui valore generalmente è -1. In molti casi la variabile esterna **int errno** è presente con un numero indicante la precisa natura dell'errore; i possibili numeri indicanti un errore sono indicati nel file **errno.h**

Solo due errori specifici occorrono nella lista alfabetica della libreria:

EDOM L'argomento passato era fuori del dominio di una funzione math

ERANGE Il risultato di un calcolo è troppo grande

L'uso della funzione di libreria **perror** o di altri simili stratagemmi è raccomandato in caso di errori inaspettati

Output formattato, (printf)

La famiglia delle chiamate **printf** è listata come

printf (Pfmt,Plst)

dove Pfmt è la stringa di controllo del formato e Plst una lista di argomenti separati da virgola. Pfmt può contenere caratteri che sono scritti letteralmente all'uscita specificata. Così:

printf("Ciao, mondo\n");

stampa la stringa a **stdout.** Pfmt può anche contenere specificazioni di formato, le quali cominciano con il segno di percentuale %, vedi tabella 6 a pag. 49 per una lista completa. Un esempio è:

char *s="Ciao";

printf ("Ciao,%s\n",s);

Per ogni specifica di formato (Δ o asterisco) in Pfmt ci dovrebbe essere esattamente un argomento dello stesso tipo in Plst, altrimenti potrebbero sorgere degli errori.

Output formattato (Scanf)

La famiglia di chiamate **scanf** è listata come

scanf (Sfmt, Slst)

dove **Sfmt** è la stringa di controllo di lunghezza e **Slst** una lista di argomenti separati da virgola, che **devono essere tutti puntatori**

Tabella 6 specifiche di formati

Una specifica di formato di stampa ha la forma
%<*flag*> <<0><*ampiezza*>><.><*precisione*><L>
«*carattere di conversione*»

flag	'—' allinea a sinistra nel suo campo l'output convertito nel suo campo '+' Δ forza un segno ('+' o '—') per conversioni segnate ' ' Δ forza un segno (' ' o '-') per conversioni segnate '#' Δ forza uno 0, 0x o 0 iniziale dove necessita.
0	specifica che la parte a sinistra deve essere riempita con 0 invece che con spazi
ampiezza	è un intero che specifica l'ampiezza minima di un campo. L'output è allineato a destra a meno di sovrascritture. Un valore convertito eccedente l'ampiezza è scritto in ogni caso separa l'*ampiezza* e la *precisione*
precisione	è una costante intera che specifica il massimo numero di caratteri in una stringa da stampare (formato s), o il numero di cifre decimali (formati e, f)
l o L	specifica che la variabile corrispondente è una **long**

Δ Alcune librerie permettono l'uso di un asterisco per modificare l'ampiezza o la precisione; in questo caso, un corrispondente **int** dovrebbe essere nella lista di output. I caratteri di conversione sono:

carattere	*argomento*	*azione svolta*
d	int	converte in formato decimale
o	int	converte in un formato ottale senza 0 iniziale
x	int	converte in un formato esadecimale senza segno e senza 0x iniziale. Cifre esadecimali possono essere in maiuscolo o minuscolo
u	int	converte in un formato decimale senza segno
c	char/int	manda in uscita un singolo carattere (caratteri NUL possono essere ignorati)

s	*char	manda in uscita come stringa di caratteri. Se la precisione è omessa oppure è 0, manda in uscita l'intera stringa fino al carattere di terminazione NUL. Se la precisione è data, manda in uscita un numero di caratteri uguali alla precisione
e	floating	converte al formato <— —> m.pppe <+>xx. Il numero delle p è determinato dalla precisione (default: 6)
f	floating	converte al formato <—> mmmm.ppp. Il numero delle p è determinato dalla precisione (default: 6)
g	floating	converte in formati d, e o f qualsiasi cosa sia più corta. Elimina gli 0 non significativi
%	—	manda in uscita il carattere %

Δ solo *i caratteri di conversione* minuscoli sono utilizzabili su tutti i sistemi.

La stringa di controllo contiene caratteri ordinari, che non siano spazi, che devono corrispondere ai caratteri che giungono dall'input stream, e le opzionali specificazioni di lunghezza che corrispondono alla lista di target opzionali in Slst; vedi tab. 7 a pag. 51. Ci deve essere esattamente un target per ogni specifica di lunghezza, eccetto il caso in cui il carattere di non assegnamento * è usato in una specifica. Spazi nella stringa di controllo, corrispondono agli spazi opzionali in input. Così

scanf ("let x=%d"&d);

corrisponde a

let x=12 oppure **letx=12**
ma non a **le tx=12**

I limiti della linea sono normalmente ignorati dal momento che il carattere di newline è un carattere di spaziatura. Le stringhe sono terminate con gli spazi, ma ciò può essere cambiato con il codice di conversione **[*scanset*]**. Lo scanset è una sequenza di

caratteri. Se il primo carattere è un accento circonflesso ^, la scanset è il set di tutti i caratteri che non sono nella susseguente sequenza di caratteri. Una serie di caratteri potrebbe essere specificata con il costrutto primo-ultimo p.e. **[0—9]**

Tabella 7 *Specifiche di scan*

Una specifica di lunghezza ha la forma:

%<><ampiezza> carattere scan*

fa si che il campo sia interpretato, ma non provoca alcun assegnamento. Nessun argomento corrispondente appare nella lista argomento.

ampiezza indica l'ampiezza massima del campo.

I possibili caratteri scan sono elencati qui sotto
Argomenti:

Carattere	*Puntatore a*	*Azione svolta*
d	int	attende un intero decimale in input
o	int	attende un intero ottale (con 0 iniziale opzionale) in input
x	int	attende un intero esadecimale (con 0x iniziale opzionale) in input
c	char	attende un carattere singolo senza omettere gli spazi. Per ottenere il primo carattere non spazio usare %1s
s	char[]	attende una stringa di caratteri. L'argomento dovrebbe essere un puntatore a un array di caratteri sufficientemente largo da contenere la stringa e il NUL che viene aggiunto
e o f	float	è atteso un numero in virgola mobile. Il formato atteso per il numero in virgola mobile è quello dato a pag. 8, a meno che sia permesso anche un segno positivo (+)

ld	long	La lettera «l» può essere usata per indicare che il corrispondente argomento è un puntatore a un elemento di tipo **long** invece che di tipo **int**, o che è un puntatore a un **double** invece che un puntatore a un **float**
lo	long	
1x	long	
le	o double	
lf	double	
hd	short	Δ alcuni sistemi usano la lettera 'h' per indicare che il corrispondente argomento punta a un elemento di tipo **short** invece che **int**
ho	short	
hx	short	
[*scanset*]	char[]	Δ è lo stesso di s (stringa) ma non vengono saltati gli spazi iniziali e inoltre la stringa può contenere solo caratteri definiti in scanset

Δ solo i caratteri scan in minuscolo sono utilizzabili su tutti i sistemi

---

Identificatori da evitare

Gli identificatori racchiusi nella tabella 8 devono essere evitati perché essi sono parole chiave o macro su alcuni compilatori. Le parole chiave sono identificate da una (K)

Tabella 8 Identificatori

abs	getchar	max
asm (k)	globaldef (k)	min
assert	globalref (k)	near (k)
auto (k)	globalvalue (k)	putc
break (k)	goto (k)	putchar
calloc	huge	readonly (k)
case (k)	if (k)	realloc
char (k)	int (k)	register (k)
clearerr	isalnum	return (k)
continue (k)	isalpha	short (k)
default (k)	isascii	sizeof (k)
do (k)	iscntrl	static (k)
double (k)	iscsym	stderr
else (k)	iscsymf	stdin
entry (k)	isdigit	stdout
enum (k)	isgraph	struct (k)
extern (k)	islower	switch (k)
far (k)	isprint	tm
feof	ispunct	toascii
ferror	isspace	tolower
fileno	isupper	toupper
float (k)	isxdigit	typedef (k)
for (k)	long (k)	union (k)
fortran (k)	mallinfo	unsigned (k)
free	malloc	void (k)
getc	mallopt	while (k)

Tabella 9 Funzioni di libreria per gruppi

generale	*conversione*	*file chiamati*	*File di stream*
abort	**atof**	**access**	**clearerr**
execl	**atoi**	**chdir**	**fclose**
exit	**atol**	**chmod**	**fdopen**
longjmp	**sprintf**	**mktemp**	**feof**
perror	**sscanf**	**unlink**	**ferror**
qsort	**swab**		**fflush**
setjmp		*file di sistema*	**fgetc**
system	*matematiche*	**close**	**fgets**
	abs	**creat**	**fileno**
memoria	**acos**	**fdopen**	**fopen**
calloc	**asin**	**fileno**	**fprintf**
free	**atan**	**lseek**	**fputc**
malloc	**atan2**	**open**	**fputs**
realloc	**ceil**	**read**	**fread**
	cos	**write**	**freopen**
carattere	**exp**		**fscanf**
isunacosa	**fabs**		**fseek**
toascii	**floor**		**ftell**
tolower	**frexp**		**fwrite**
toupper	**hypot**		**getc**
	ldexp		**getw**
stringa	**log**		**putc**
index	**log10**		**putw**
rindex	**modf**		**rewind**
sprintf	**pow**		**setbuf**
sscanf	**rand**		**ungetc**
strcat	**sin**		
strchr	**sqrt**		*standard*
strcmp	**srand**		*output*
strcpy	**tan**		**getchar**
strlen	**tanh**		**gets**
strncat			**printf**
strncmp			**putchar**
strncpy			**puts**
strrchr			**scanf**

Lista alfabetica delle funzioni di libreria

void **abort** () library function
Δ KR CR DR LAT DES
ABORT forza la fine di un processo, generalmente per scopi di debugging. Se possibile, tutti i file aperti sono chiusi e, viene generato un 'core dump'. La funzione **abort** generalmente non ritorna alcun valore.

int **abs**(x) library macro stdio.h
int x=-123; Δ KR CR CI
ABS ritorna il valore assoluto del suo argomento intero.
Alcune volte è implementata come macro

 # define abs (x) ((x)<0? (—(x)):(x))

in stdio.h. Per le variabili in virgola mobile si usa la **fabs.** La vostra libreria potrebbe comprendere anche la funzione equivalente **labs** per interi lunghi.
Il valore di ritorno della funzione potrebbe non essere corretto per il minimo intero negativo per esempio, se la dimensione di un intero è 8 bit, **abs (—128)** potrebbe ritornare -128. ALcune implementazioni catturano questo errore.

int **access** (Fpath,amode) system function
int amode=2; Δ KR CR CI CC LAT DES
ACCESS controlla l'accessibilità del file chiamato. Se **amode** è 0, Δ viene controllata l'esistenza del file, altrimenti **amode** ha il seguente significato:
1 Δ esecuzione (ricerca, se path è una directory)
2 scrittura
4 lettura
la funzione ritorna uno 0 se l'accesso richiesto è permesso; altrimenti ritorna -1 e indica in **errno** il codice di errore appropriato.

double **acos**(r) math function math.h
double r; Δ KR CR DR LAT

ACOS ritorna l'arcocoseno di r che è misurato in radianti. Se r non è compreso nell'intervallo [-1...1] **acos** ritorna 0 e pone il valore di **errno** a EDOM. Il valore ritornato è compreso tra 0 e π.

double **asin**(r) math function math.h
double r; Δ KR CR DR LAT
ASIN ritorna l'arcoseno di r, che è misurato in radianti. Se r non è compreso fra [-1...1], **asin** ritorna zero ponendo il valore di **errno** a EDOM. Il valore calcolato è compreso tra $-\pi/2$ e $\pi/2$.

double **atan**(r) math function math.h
double r; Δ KR CR DR LAT
ATAN ritorna l'arcotangente di r che è misurato in radianti. Il valore calcolato è compreso tra $-\pi/2$ e $\pi/2$.

double **atan2**(x,y) math function math.h
double x,y; Δ KR CR DR LAT DES
ATAN2 ritorna l'arcotangente di y/x. Il valore calcolato è compreso tra $-\pi$ e π. Il segno di entrambi gli argomenti è utilizzato per determinare il quadrante. Se x è zero, zero è il valore calcolato e Δ il valore di **errno** è EDOM.

double **atof**(s) library function —
char *s; Δ CR LAT
ATOF converte la stringa di caratteri s in una **double.** La conversione finisce con il primo carattere non riconosciuto.
Spazi bianchi possono precedere la rappresentazione in caratteri del **float.** Il formato di float è dato a pag. 7, ma inoltre è possibile utilizzare un segno positivo (+). Non compaiono indicazioni di errori o di casi di overflow.

int **atoi**(s) library function —
char *s; Δ CR LAT
ATOI converte la stringa di caratteri s in un **int.** la conversione finisce con il primo carattere non riconosciuto. Lo spazio può precedere il carattere rappresentazione dell'**int.** Non compaiono indicazioni di errori o di casi di overflow.

long **atol**(s) library function —
char *s; Δ KR CR CI LAT
ATOL è identica a **atoi,** ma converte le stringhe in **long**

char ***calloc**(number,size) system macro malloc.h
int number,size; Δ CR
CALLOC alloca un'area di memoria, sufficiente a contenere la registrazione del numero **number** di elementi di dimensioni **size** e lo inizializza a 0. Il puntatore di ritorno può contenere un elemento di qualsiasi tipo. Se non c'è memoria sufficiente a disposizione, in uscita compare il puntatore NULL. La funzione **calloc** è tipicamente utilizzata con operatori **sizeof** come in

p=calloc (20, sizeof (int));

Una versione più veloce (macro) di **calloc** può trovarsi in malloc.h.

double **ceil**(x) math function math.h
double x; Δ KR CR DR LAT DES
CEIL ritorna (come **double**) il più piccolo intero che non sia minore dell'argomento (simile alla **floor)**

int **chdir**(Dpath) system function —
Δ KR CR CC DR LAT DES
CHDIR cambia la working directory attualmente funzionante. Uno 0 è ritornato se la directory è stata cambiata correttamente, altrimenti compare un 1. La working directory è la directory usata per quei file il cui nome non comincia con il carattere /. Questa funzione ha senso solo su sistemi con una struttura di directory ad albero.

int **chmod**(Fpath,pmode) system function —
int pmode=0600 Δ KR CR CI CC LAT DES
CHMOD cambia la protezione di un file. Se l'operazione è conclusa correttamente viene mandato in uscita uno 0, altrimenti un -1. L'utente deve avere il permesso di scrittura per il file. Su

sistemi UNIX, **pmode** è una bit map che descrive i seguenti permessi:

0400 owner, lettura	0040 group, lettura	0004 world, lettura
0200 owner, scrittura	0020 group, scrittura	0002 world, scrittura
0100 owner, esecuzione	0010 group, esecuzione	0001 world, esecuzione

Un privilegio di scrittura implica un privilegio di cancellazione

void **clearerr**(StreamP) library macro stdio.h
Δ KR CR LAT DES
CLEARERR azzera le indicazioni di errore per StreamP, così che **ferror** cessa di indicare un errore. Le indicazioni di errore persistono finché la **clearerr** non agisce o lo stream viene chiuso.

int **close**(Handle) system function —
Δ CC
CLOSE chiude il file associato con l'Handle. Se il file era aperto per scrittura, qualsiasi carattere che si trovi nel buffer viene prima mandato in uscita. La funzione ritorna uno 0 se ha chiuso il file correttamente altrimenti compare un -1. Δ La causa di un errore in questa funzione potrebbe essere resa nota da **errno**

double **cos**(r) math function math.h
double r; Δ KR CR LAT
COS ritorna il coseno del suo argomento che viene misurato in radianti. Il valore ritornato è compreso tra [-1...1]

int **creat**(Fpath, Pmode) system function —
int pmode=0400 Δ CC
CREAT crea un nuovo file. Se esiste già un file con lo stesso nome, potrebbe essere troncato. La funzione **creat** ritorna un **int** piccolo e positivo (l'Handle) se può creare il file correttamente, altrimenti manda in uscita un -1. Se il file è creato, il **pmode** determina i permessi di accesso (vedi **chmod**).
Bisogna essere preparati a un aggravarsi della situazione su sistemi che richiedono una distinzione tra file tradotti («translated»,

cioè file di testo) e file in formato binario (non tradotti). Si potrebbe avere, per supplire a ciò, un terzo argomento, un bit indicatore una traduzione **pmode,** una funzione con un nome differente, una variabile **extern** separata o una differente procedura di linking per creare o aprire un file binario. Oppure potrebbe essere completamente impossibile creare un file con il modo desiderato.

int **execl**(Fpath,arglist, (char*)0) system function —
Δ KR CR CI CC LAT DES
EXECL esegue (concatena) il programma specificato da Fpath. Se tutto è corretto la funzione non ha ritorno e il controllo passa al programma specificato. Se l'operazione non ha successo la funzione manda in uscita un -1 e **errno** indica i motivi del mancato funzionamento. Dove applicabile il processo concatenato eredita la situazione originale. Arglist consiste di zero o più puntatori a stringhe di argomenti che sono passati alla funzione **main** del processo da concatenazione come in

execl ("cat", "file.1", "file.2", (char *)0);

void **exit**(status) standard function —
int status=0
EXIT termina il processo dal quale è stato chiamato. Lo **status** specificato (che altrimenti potrebbe essere omesso) è passato al processo chiamante (per esempio la shell o l'interprete dei comandi). Per convenzione, il ritorno di uno 0 significa che l'operazione è stata eseguita correttamente; ogni altro valore in ritorno significa errore. Ogni file è prima chiuso; **_exit** evita queste azioni di chiusura. Se si va oltre i limiti della funzione **main** è come se si chiamasse **exit (0).**

double **exp**(p) math function math.h

double p; Δ KR CR LAT
EXP ritorna la costante **e** elevata alla potenza dell'argomento p. Se avviene un caso di overflow, **exp** potrebbe ritornare un valore enorme e Δ porre **errno** a ERANGE. Questa funzione potrebbe essere scritta come

pow (2.71828182845905, p)

double **fabs**(x) math function math.h
double x; Δ KR CR LAT
FABS ritorna il valore assoluto di x. Alcuni sistemi permettono che la macro **abs** sia usata con questo scopo (non è consigliabile a meno che non la definiate da voi)

int **fclose**(StrasmP) standard function stdio.h
FCLOSE chiude il file StreamP. Ogni buffer associato viene svuotato e i buffer allocati per l'I/O vengono disallocati.
Il valore di ritorno è 0 se il file è stato chiuso correttamente, altrimenti è EOF. I file dovrebbero venire chiusi anche se essi sono aperti solo per lettura. Tutti i file aperti vengono chiusi automaticamente quando si chiama la **exit.**

FILE ***fdopen**(Handle,rwa) library function stdio.h
char *rwa Δ KR CR CI CC LAT DES
FDOPEN associa un puntatore stream con un Handle. Ciò permette l'uso di funzioni come **putc** che accede a un file precedentemente aperto con chiamate come **open.** Il primo argomento è lo Handle ottenuto con una **open** o una **creat;** il secondo è un puntatore a una stringa analoga all'argomento **rwa** della **fopen;** essa deve concordare con il **mode** con cui il file è stato aperto. L'uso di questa funzione dovrebbe essere evitato. (confronta **fileno).**

int **feof**(StreamP) standard macro stdio.h
Δ CR CI DES
FEOF ritorna un valore diverso da 0 se la fine file è stata letta su StreamP. Uno 0 in uscita indica che la fine non è stata raggiunta.

int **ferror**(StreamP) standard macro stdio.h
Δ CR DES
FERROR controlla se si è verificato un errore in fase di lettura o in fase di scrittura. Uno 0 in uscita indica che non ci sono stati errori. Qualsiasi altro valore indica la presenza di un errore. L'indicatore

di errore persiste finché è cancellato con la **clearerr** o viene chiuso lo stream. Questa macro è molto utile per controllare la correttezza di funzioni come **getw** che non possono indicare casi di errore con affidabilità.

int **fflush**(StreamP) standard function stdio.h
Δ CR DES

FFLUSH trasferisce ogni informazione messa su buffer nello StreamP. I file in uscita sono normalmente su buffer a meno che non siano collegati a un terminale. Se vi è un errore la funzione ritorna un EOF.

int **fgetc**(StreamP) standard function stdio.h
Δ CR

FGETC ritorna il prossimo carattere dello StreamP; essa può usare **malloc** per allocare un buffer se questa è la prima richiesta di I/O per StreamP. La funzione dà in uscita un EOF in caso di fine file o errore. Caratteri da un terminale non sono normalmente disponibili finché non si raggiunge la fine della linea. È consigliabile utilizzare **fgetc** piuttosto che **getc** se la lunghezza del programma è importante.

char ***fgets**(b,maxb,StreamP) standard function stdio.h
char b[100];
int maxb=100;
FGETS legge una linea dello stream StreamP. Essa legge maxb-1 caratteri o si ferma al primo carattere newline. Nell'ultimo caso, b includerà il carattere newline (\n) Se si tenta la lettura dopo la fine del file o se vi è stato un errore, **fgets** potrebbe ritornare il puntatore NUL; altrimenti ritornerà il primo argomento.

int **fileno**(StreamP) standard macro stdio.h
Δ CR CI CC DES

FILENO ritorna il numero intero Handle associato con lo specifico puntatore stream. Bisogna stare attenti, dal momento che molti sistemi non permettono l'accesso allo stesso file da Streamp e da Handle.

double **floor**(x) math function math.h
Δ KR CR DR LAT DES

FLOOR ritorna, come **double,** il più grande intero non maggiore dell'argomento. (Simile alla **ceil.**)

FILE ***fopen** (Fpath, rwa) standard function stdio.h
char *rwa;

FOPEN apre un file e ritorna un puntatore allo stream associato. Se **fopen** non può aprire il file, manda in uscita il puntatore NULL. Tipicamente nessun buffer è allocato in questa fase; i buffer sono allocati dalla prima chiamata a **getc** o **putc.**
La modalità di accesso è specificata dalla **rwa** ed è ufficialmente una delle seguenti:
"r" Apre un file per lettura. Il file è posizionato all'inizio del file
"w" Apre un file solo per scrittura. Se non esiste il file viene creato. Se esiste un file con lo stesso nome può venire troncato.
"a" Apre un file per aggiungere. È quindi simile alla "w" ma in questo caso i dati sono aggiunti al termine del file.
È illegale leggere da file che sono aperti in modo "w" oppure "a" cosi come non si può scrivere in un file aperto in modo "r". Alcuni sistemi posseggono anche altri modi:
Δ"r+" Apre un file in lettura e si posiziona all'inizio del file, ma è possibile anche scriverci
Δ"w+" È un file di scrittura, ma si può anche leggervi
Δ"a+" È un file in cui aggiungere dati, ma si può anche leggervi
Dove la lettura e la scrittura sono permesse un **fseek** o **rewind** è richiesto tra le differenti operazioni.
Inoltre dei sistemi che richiedono traduzione dei file possono avere anche più modi d'accesso che possono essere combinati insieme:
Δ "rb" o "r+b" apre un file per lettura non tradotta
Δ "wb" o "w+b" apre un file per scrittura non tradotta
Inoltre, sono anche usati altri metodi per indicare il tipo di accesso

int **fprintf** (StreamP,Pfmt,Plst) standard function stdio.h
FPRINTF opera un output formattato verso il file StreamP. La funzione ritorna il numero di caratteri realmente scritti; se compare un -1 significa che vi è stato un errore. Vedi pag. 48, 49 per i dettagli su questi argomenti.

int **fputc** (c, StreamP) standard function stdio.h
char c; Δ CR
FPUTC scrive un singolo carattere a StreamP; questa funzione può utilizzare **malloc** per allocare un buffer se questa è la prima richiesta di I/O per il file StreamP. La funzione ritorna il carattere o l'EOF in caso di errore. Si consiglia di usare **fputc** invece di **putc** se la lunghezza del programma è importante.

void **fputs**(s,StreamP) standard function stdio.h
char *s;
FPUTS scrive la stringa s allo stream specificato. La funzione non aggiunge un newline (\n) alla stringa. Non si hanno valori di ritorno. Questa funzione è equivalente a

fprint (streamP, "%s",s)

int **fread**(p,bytes,numero,StreamP)
standard function stdio.h
char *p; Δ CR
FREAD legge il **numero** specificato di item di una lunghezza data **(bytes)** nella memoria cominciando da p. Gli item sono letti dallo stream StreamP, e viene riportato il numero di item realmente letti.
EOF o 0 compaiono in caso di errore; in questo caso l'ultimo item potrebbe essere incompleto. Lo stream, normalmente deve essere aperto in modo binario. Questa funzione è utilizzata per leggere item scritti con **fwrite**

void **free**(m) library macro malloc.h
char *m;
FREE libera un area di memoria precedentemente allocata con **malloc, calloc** o **realloc.** Una versione più veloce (macro) di **free** potrebbe essere in malloc.h.

FILE ***freopen** (Fpath,rwa,StreamP)   standard function stdio.h
char *rwa;   Δ CR CI DES
FREOPEN chiude lo stream associato con StreamP e successivamente sostituisce il file specificato da Fpath, p. e. essa esegue **fopen (Fpath, rwa).** La funzione ritorna il suo terzo argomento (il puntatore stream) se tutto è corretto, altrimenti ritorna il puntatore NULL. Un tipico uso è di associare uno dei nomi predefiniti **stdin, stdout, stderr** con un file, come in
if (freopen ("out. log", "a", stdout))...
Poiché **stdout** è una costante, una combinazione di **fclose** e **fopen** non potrebbe essere usata, a meno che si fosse assunto che una **fopen** ritorna automaticamente il puntatore stream rilasciato dall'ultimo **fclose**

double **frexp**(x,pi)   library function —
double x;   Δ KR CR DR LAT DES
int *pi;
FREXP separa un numero in virgola mobile in mantissa e esponente. Essa ritorna la mantissa nell'intervallo [0.5...1.0).
L'esponente è memorizzato in un int puntato da **pi.** Perciò se
mant=frexp (x,pi)
poi
mant*2.0(*pi)
è il valore originale.

int **scanf** (StreamP,Sfmt,Slst)   standard function stdio.h
FSCANF esegue input formattati provenienti dallo StreamP. La funzione ritorna un EOF se la fine file è stata raggiunta; altrimenti ritorna il numero di operazioni riuscite e di item convertiti. Vedi pag. 48, 49 per dettagli su questo argomento.

int **fseek**(StreamP,off,whence)   standard function stdio.h
long off;
int whence;
FSEEK posiziona lo stream StreamP in uno specificato byte (offset) nel file. Se ritorna uno 0 l'operazione è stata svolta correttamente, se ritorna un EOF significa che vi è stato un errore. L'offset **off** è misurato da una locazione chiamata **whence** che potrebbe essere

0 dall'inizio del file
Δ 1 dalla posizione attuale di lettura o scrittura
Δ 2 dalla fine del file
Questa funzione potrebbe non funzionare correttamente su file-system di file non UNIX; su alcuni sistemi, **fseek** potrebbe essere usato solo per ricercare locazioni assolute (**whence=0**) precedentemente ritornate da **ftell.** La **fseek** distrugge qualsiasi carattere «mandato indietro» con **ungetc.** Il risultato è indefinito su periferiche non ad accesso diretto (terminali, stampanti).

long **ftell**(StreamP) standard function stdio.h
Δ CR DES
FTELL ritorna la posizione del byte attuale dello stream file specificato misurato dall'inizio del file. In caso di errore si ottiene un -1L. Su sistemi con file translation il valore di ritorno è utile solo per **fseek.**

int **fwrite**(p,bytes,number,StreamP) standard function stdio.h
char *p: Δ CR
int bytes, number;
FWRITE scrive un numero, specificato da **number,** di items di ampiezza uguale a **bytes** allo stream StreamP, cominciando dalla locazione p. La funzione ritorna il numero di byte attualmente scritti oppure 0 in caso di errore. La funzione **fread** è normalmente usata per leggere item scritti con la funzione **fwrite.**

int **getc** (StreamP) standard macro stdio.h
GETC ritorna il prossimo carattere dallo StreamP; la funzione può utilizzare **malloc** per allocare un buffer se questa è la prima richista di I/O dallo StreamP. In caso di fine file o errore manda in uscita l'EOF. È meglio utilizzare **getc** piuttosto che **fgetc** se la velocità è importante.

int **getchar**() standard macro stdio.h
GETCHAR è equivalente a **getc(stdin).** I caratteri da un terminale non sono generalmente disponibili fino alla fine della riga (end-of-line).

char ***gets**(b) library function stdio.h
char b[100]; Δ KR CR CI
GETS legge una linea dallo stream standard di input e la pone in b. Un NUL è aggiunto alla linea; un newline al termine non è incluso. Il buffer b deve essere abbastanza ampio da contenere la stringa. Un puntatore NULL compare in caso di errore o di fine file altrimenti **gets** ritorna il suo argomento

int **getw**(StreamP) standard function stdio.h
Δ CR LAT
GETW legge un **int** binario da StreamP e ritorna quell'**int.** Il numero esatto di byte letti dipende dalla lunghezza di un **int** sul sistema utilizzato. In caso di errore compare un EOF ma potrebbero anche venire utilizzati **int,ferror** o **feof** per segnalare un errore. Dove è possibile, StreamP deve essere stato aperto in modo binario. La funzione **getw** è utilizzata per rileggere un valore scritto da putw. Funzioni equivalenti **getl** e **putl** possono essere usate per interi tipo **long.**

double **hypot** (x,y) math function math.h
double x,y; Δ KR CR CI DR LAT DES
HYPOT ritorna il risultato di **sqrt (x*x+y*y)**

char ***index**(s,c) library function —
INDEX è identica a **strchr** che è preferibile

int **is*unacosa***(char) standard macro ctype.h
int **isalnum**(c) Δ CR
int **isalpha**(c)
int **isascii**(c) Δ CR
int **iscntrl**(c) Δ CR
int **isdigit**(c)
int **islower**(c)
int **isprint**(c) Δ CR
int **ispunct**(c) Δ CR
int **isspace**(c)
int **isupper**(c)
IS*UNACOSA* sono macro che ritornano numeri interi diversi da 0

se l'argomento appartiene a una certa classe di caratteri. Il valore di ritorno di **isascii** è definito per qualsiasi **int.** Tutte le altre macro sono definite solo se **isascii(c)** è vero, a meno che **c** è EOF. La tabella 10 illustra alcune di queste macro; avvertiamo che esistono casi in cui sono presenti piccole variazioni.

Tabella 10 *Le macro ISUNACOSA*

Intervallo	*alnum*	*alpha*	*ascii*	*cntrl*	*lower*	*print*	*punct*	*space*	*upper*
0...8	○	○	●	●	○	○	○	○	○
0x9(\t)	○	○	●	●	○	○	○	○	○
0xa (\n)	○	○	●	●	○	○	○	●	○
0xb (\v)	○	○	●	●	○	○	○	●	○
0xc (\f)	○	○	●	●	○	○	○	●	○
0xd (\r)	○	○	●	●	○	○	○	●	○
0xe—0x1f	○	○	●	●	○	○	○	○	○
Spazio	○	○	●	○	○	●	○	●	○
!"#$%&()*+,-./	○	○	●	○	○	●	●	○	○
'0'...'9'	●	○	●	○	○	●	○	○	○
:;<=>?@	○	○	●	○	○	●	●	○	○
'A'...'Z'	●	●	●	○	○	●	○	○	●
[\]^-'	○	○	●	○	○	●	●	○	○
'a'...'z'	●	●	●	○	●	●	○	○	○
{|}~	○	○	●	○	○	●	●	○	○
0x7f (DEL)	○	○	●	●	○	○	○	○	○
0x80...	?	?	○	?	?	?	?	?	?
EOF	○	○	○	○	○	○	○	○	○

double **ldexp**(x,y) library function math.h
double x; Δ KR CR DR LAT DES
int y;
LDEXP ritorna x moltiplicato per 2 alla potenza di y. Può essere, quindi, scritta come **x * pow (2.0, (double)y).** Errori possono essere causati da overflow o underflow e Δ in tal caso viene assegnato ad **errno** il valore ERANGE. (confronta con **frexp**)

double **log**(x) math function math.h
double x; Δ KR CR LAT

LOG ritorna il logaritmo naturale (base e) dell'argomento. Se x è 0 o negativo, il valore risultante è indefinito e Δ **errno** viene posto a EDOM

double **log10**(x) math function math.h
double x; Δ KR CR LAT
LOG10 ritorna il logaritmo in base 10 dell'argomento. Se x è 0 o negativo, il valore risultante è indefinito e Δ **errno** viene posto a EDOM.

int **longjmp** (env, retv) library function setjmp.h
jmp-buf enr;
int retr;

LONGJMP riporta all'ambiente di elaborazione memorizzato dall'ultima chiamata a **setjmp.** Ritorna a **setjmp** come se quella funzione fosse appena stata chiamata e avesse riportato il valore **retv.** Vedi **setjmp** (pag. 73) per averne un esempio.

long **lseek**(Handle,off,whence) system function —
long off; Δ CC
int whence;
LSEEK posiziona un file in una posizione arbitraria misurata in byte (**off**) da:
  l'inizio file se **whence** è 0
Δ la posizione corrente se **whence** è 1
Δ la fine del file se **whence** è 2
La funzione ritorna la nuova locazione del puntatore (misurata dall'inizio del file), o -1L se è stata tentata una seek illegale. Operazioni di seek dopo la fine del file forniscono risultati inattendibili. Richieste di seeking a altre posizioni oltre quelle ottenute usando precedentemente la funzione lseek sopra sono non affidabili nel caso di file tradotti.

char ***malloc**(bytes) standard macro malloc.h
unsigned bytes; Δ CR
MALLOC alloca una area di memoria di ampiezza uguale a **bytes.** La funzione ritorna un puntatore al primo byte che può puntare a qualsiasi elemento.

L'area non è inizializzata. Se non c'è spazio sufficiente viene ritornato un puntatore NULL.

char ***mktemp**(template) library function —
char template[]="krXXXXXX"; Δ KR CR CI CC LAT DES
MKTEMP genera un unico filename a partire da uno schema. Le sei X sono sostituite da una serie unica di caratteri. La funzione sostituisce il nome del file allo schema e ritorna il puntatore allo stesso; il puntatore punterà a una stringa vuota se un unico filename non può essere creato.

double **modf**(x,intptr) library function —
double x; Δ KR CR DR LAT DES
double *intptr;
MODF ritorna la parte frazionaria positiva di x e memorizza la parte intera tramite **intptr.**

int **open**(Fpath,mode) system function —
int mode

OPEN apre un file e ritorna un Handle. Il **mode** è uno dei seguenti:
0 apre un file per lettura
1 apre un file per scrittura
2 apre un file per aggiornamento (lettura e scrittura).
Se il file non può non essere aperto, **open** ritorna -1 altrimenti il valore di ritorno è un intero piccolo e positivo che è un descrittore del file. Il file è posizionato all'inizio file. Questa funzione ha alcuni difetti come la funzione **creat** (vedere p. 58).

void **perror**(s) library function —
char *s="oops"; Δ KR CR CI CC LAT DES
PERROR manda in uscita un breve messaggio di errore a **stderr** chè descrive l'ultimo errore incontrato durante una chiamata alla libreria. L'ultimo errore è ottenuto da **perror.** La funzione **errno** è equivalente a **fprint (stderr, "%s:%s\n",s,sys_errlist [errno])**

double **pow**(x,y) math function math.h
double x,y; Δ KR CR DR LAT

POW ritorna **x** elevato alla potenza di **y.** Un numero molto grande è restituito in caso di overflow, e quindi Δ **errno** è posto a ERANGE. Se x e y sono contemporaneamente 0, **pow** ritorna uno 0. Se y è negativo e non intero il risultato è indefinito e Δ **errno** è posto a EDOM.

int **printf**(Pfmt,Plst) standard function stdio.h
PRINTF fornisce un output formattato sullo standard output. La funzione ritorna il numero di caratteri effettivamente stampati: se il valore di ritorno è -1 significa che vi è stato un errore. Vedi pag. 48, 49 per i dettagli su questo argomento.

int **putc**(c,StreamP) standard macro stdio.h
char c;
PUTC scrive un singolo carattere a StreamP; tale funzione utilizza **malloc** per allocare un buffer nella prima chiamata di I/O allo StreamP. La funzione ritorna il carattere oppure in caso di errore, l'EOF. Utilizzare **putc** in preferenza di **fputc** se la velocità è importante.

int **putchar**(c) standard macro stdio.h
char c;
PUTCHAR è equivalente a **putc (c,stdout)**

int **puts**(s) library function stdio.h
char *s; Δ KR CR
PUTS scrive una stringa di caratteri (escludendo il finale NUL) a **stdout,** seguito da un newline. La funzione è equivalente a **printf ("%s\n", s)**

int **putw**(x,StreamP) standard function stdio.h
int x; Δ CR LAT
PUTW scrive un numero di caratteri che sono la rappresentazione binaria di x allo stream StreamP. La funzione ritorna EOF in caso di errore ma, dal momento che EOF è un normale numero intero, gli errori dovrebbero essere controllati da **ferror.** Lo stream deve essere binario. La funzione **getw** dovrebbe essere usata per leggere l'**int** dallo stream. I file creati con **getw** non sono trasferibili tra sistemi.

void **qsort**(base,nel,bytes,compar) library function —
unsigned nel; char *base; Δ KR CR VAX MC LAT
int bytes;
int (*compar)();

QSORT implementa un algoritmo di tipo quicksort per ordinare un vettore di **nel** elementi a partire da **base.** L'ampiezza di ogni elemento è uguale a **bytes.** La comparazione è eseguita dalla funzione puntata da **compar;** la funzione di comparazione utilizza due puntatori ai due elementi da comparare e deve ritornare:

0 se il primo è == al secondo
-1 se il primo è < del secondo
1 se il primo è > del secondo

L'esempio seguente ordina un array di puntatori:

```
char * aps [ ]={"ciao", "mondo", "arrivederci", "mondo"};
int i;
compar (a,b)
char **a, **b;
{
	return (strcmp (*a, *b) );
}
main()
{
	qsort(aps,4,sizeof(char *),compar);
	for (i=0;i<4;i++)
	  printf("%s",aps[i]);
}
```

Il risultato è **arrivederci ciao mondo mondo** sull'output.

int **rand**() library function —
Δ KR CR CI LAT

RAND ritorna un numero positivo pseudocasuale di tipo **int.** È imprudente fare assegnamento sulla distribuzione o il periodo dei numeri generati. (Vedi anche **srand).**

int **read**(Handle,b,bytes) system function —
char b[100]; Δ CC
int bytes=100;

READ tenta di leggere il numero specificato di byte dal file. Il valore di ritorno equivale al numero di byte letti in realtà. Quest'ultimo valore potrebbe essere minore di **bytes** se l'input giunge da un terminale (solo una linea alla volta può essere letta) o se il file è tradotto. Un valore di ritorno uguale a 0 significa fine file. EOF significa che si è verificato un errore.

char ***realloc**(p,bytes) library macro malloc.h
char *p; Δ KR CR LAT
int bytes;
REALLOC cambia l'ampiezza dell'area puntata da p, nel numero di byte specificato da **bytes.** Essa ritorna un puntatore alla nuova area. Inoltre la funzione è libera di spostare l'area originale, così puntatori e elementi interni all'area potrebbero risultare sbagliati.

int **rewind**(StreamP) standard function stdio.h
Δ CR CI
REWIND resetta la posizione di lettura, scrittura per lo stream StreamP all'inizio del file. La funzione ritorna EOF in caso di errore. Questa funzione è identica a **fseek (StreamP,0,L,0).** Qualsiasi carattere «mandato indietro» con **ungetc** è cancellato.

rindex(s,c) library function —
RINDEX è identica a **strrchr** che è preferibile.

int **scant**(Sfmt,Slat) standard function stdio.h
SCANF esegue input formattati dallo standard input **stdin.** La funzione ritorna EOF se la fine file è stata letta; altrimenti i valori di ritorno indicano il numero di input eseguiti correttamente e gli item convertiti. Vedi pag. 48, 49 per i dettagli su questo argomento.

setbuf(StreamP,buf) standard function stdio.h
char *buf=NULL Δ CR CI DES
SETBUF associa il buffer puntato da **buf** con lo stream StreamP. Se lo StreamP è NULL il file sarà senza buffer. Altrimenti la dimensione del buffer dovrà essere di ampiezza BUFSIZ (una costante

definita in stdio.h) o più grande. Questa funzione dovrebbe essere usata dopo che un file è stato aperto, ma prima che venga eseguito qualsiasi operazione di I/O.

int **setjmp**(env) library function setjmp.h
jmp-buf env; Δ KR CR CI LAT
SETJMP rende possibile un goto non locale salvando l'ambiente di elaborazione della funzidone chiamante in un aggregato il cui tipo **(jmpbuf)** è definito in <setjmp.h>. La funzione ritorna 0 alla prima chiamata. Sembrerà richiamata di nuovo in caso venga chiamata **longjmp.** L'esempio seguente illustra il suo uso:

```
# include <setjmp.h>
jmp-buf savenv;
main ()
{
    switch (setjmp (savenv) )
    {
        case 0: submarin ();
                break;
        case 1: printf ("Can't find water\n");
                break;
    }
}
submarin ()
{
    FILE *fp;
    if (fp=fopen ("water.fil", "r") )
    {
    /* read water.fil,then */
    fclose (fp).
    }
    else
    }
    /* could not open water.file */
    longjmp (savenv,1);
```

double **sin**(r) math function math.h
double r; Δ KR CR LAT
SIN ritorna il seno di r che è misurato in radianti. Il valore di ritorno è compreso tra [-1...1]

int **sprintf**(s,Pfmt,Plst) standard function —
char s[100];
SPRINTF esegue output formattati in array di carattere. La funzione eventualmente ritorna il numero di caratteri realmente mandati in uscita (escludendo il NUL terminante). Un -1 in ritorno indica la presenza di un errore. Vedi pag. 48, 49 per i dettagli su questo argomento.

double **sqrt**(x) math function math.h
double x; Δ KR CR LAT
SQRT ritorna la radice quadrata del suo argomento. Se x è negativo uno 0 è mandato in uscita e Δ **errno** è posto a EDOM

void **srand**(seed) library function —
int seed; Δ KR CR CI CC LAT
SRAND inizializza il generatore di numeri casuali **rand** con lo starting point **seed.** Se **seed** è un numero dipendente dall'implementazione (come 1 o 5), **srand** reinizializza il generatore di numeri casuali.

int **sscanf**(s,Sfmt,Slat) standard function stdio.h
char *s;

SSCANF esegue input formattati dalla stringa s. La funzione ritorna un EOF se il NUL terminante è stato letto; altrimenti ritorna il numero di item sui quali le operazioni di riconoscimento e conversione sono state effettuate con successo. Vedi pag. 48, 49 per i dettagli su questo argomento.

char ***strcat**(d,s) library function string.h
char *d,*s; Δ CR
STRCAT aggiunge la stringa s alla stringa d. Entrambe le stringhe devono terminare con il valore NUL. La funzione ritorna il suo primo argomento. La stringa d deve essere abbastanza grande da

contenere il risultato. La chiamata **strcat**(d,d) può causare la perdita della stringa.

char ***strchr**(s,c) library function string.h
char *s="find me"; Δ KR CR LAT
char c='m';
STRCHR cerca il primo carattere uguale a c nella stringa s. La funzione ritorna il puntatore NULL se il carattere non è stato trovato; altrimenti ritorna un puntatore alla posizione del primo carattere uguale a c. (La funzione **index** è identica.)

int **strcmp**(s1,s2) library function string.h
char *s1,*s2 Δ CR
STRCMP confronta 2 stringhe. La funzione ritorna un intero che rappresenta il confronto lessicografico delle due stringhe ed è:
negativo se s1 è minore o più corta di s2
0 se s1 è uguale a s2
positivo se s1 è maggiore o più lunga di s2
Questa funzione può usare il confronto di caratteri con segno

char ***strcpy**(db,s) library function string.h
char db[100];
char *s=" to be copied";
STRCPY copia la stringa s a **db.** La funzione si ferma dopo che ha copiato il NUL terminante di s. Il buffer **db** deve essere abbastanza grande per contenere il risultato. La funzione ritorna il primo argomento.

int **strlen**(s) library function string.h
char *s; Δ CR
STRLEN ritorna un valore pari alla lunghezza della stringa s, escludendo il carattere NUL di fine stringa.

chr ***strncat**(d,s,mx) library function string.h
char *d,*s; Δ KR CR LAT
int mx;
STRNCAT aggiunge la stringa s alla stringa d, utilizzando fino a **mx** caratteri della stringa s. La stringa d deve quindi esistere e la

stringa s dovrebbe terminare con NUL. La stringa d deve essere abbastanza grande da contenere d e s. La funzione ritorna il suo primo argomento.

int **strncmp**(s1,s2,mx) library function string.h
char *s1,*s2; Δ KR CR LAT
int mx;
STRNCMP confronta due stringhe. La funzione ritorna un intero che rappresenta il confronto lessicografico (vedi **strcmp**).
L'argomento **mx** dà il massimo numero di caratteri da confrontare. Questa funzione può usare il confronto di caratteri con segno.

char ***strncpy**(db,s,mx) library function string.h
char db[100]; Δ KR CR LAT
char *s="Ciao Mondo"; int mx=100;
STRNCPY copia la stringa s in d, fino al numero di caratteri specificato da **mx.** Il primo argomento viene ritornato dalla funzione. Se **mx** caratteri sono copiati, il risultato non terminerà con il carattere NUL.

strrchr(s,c) library function string.h
char *s; Δ KR CR LAT
char c;
STRRCHR ritorna un puntatore all'ultimo carattere uguale a c nella stringa s terminante con un NULL. Se il carattere non viene trovato viene ritornato il puntatore NUL. (Identica a **rindex**)

void **swab**(s,d,bytes) library function
char *s,sd; Δ KR CR VAX CI LAT DES
int bytes;
SWAB copia il numero di byte indicato da **bytes** dal sorgente s alla destinazione d, scambiando byte pari e byte dispari

int **system**(s) standard function
char *s="ps -l"; Δ VAX CI DR LAT DES
SYSTEM causa la chiamata di una copia della shell come se la stringa s fosse stata introdotta da un terminale. Il valore di ritorno

è il valore ritornato dalla shell; esso è diverso da 0 se si è verificato un errore. Benché faccia parte della libreria standard di I/O, questa funzione non si trova facilmente su sistemi che non siano UNIX.

double **tan**(r) math function math.h
double r; Δ KR CR LAT U7
TAN ritorna la tangente del suo argomento che è misurato in radianti. Un valore enorme viene ritornato se r ha un valore particolare (...$-3\pi/2$, $-\pi/2$, $\pi/2$, $3\pi/2$...) e Δ **errno** è posto a ERANGE

double **tanh**(r) math function math.h
double r; Δ KR CR CI DR LAT DES
TANH ritorna la tangente iperbolica del suo argomento che è misurato in radianti

int **toascii**(c) standard macro ctype.h
int c; Δ KR CR CI CC LAT DES U7
TOASCII trasforma il carattere o intero c in un carattere ASCII compreso tra 0...0x7f. La funzione non è utilizzabile su macchine che non usano il set di caratteri ASCII. Rovina il valore EOF, il quale dovrà essere verificato prima di usare **toascii.**

int **tolower**(c) standard macro ctype.h
char c;
TOLOWER ritorna il carattere minuscolo equivalente al carattere maiuscolo c. Il risultato è indefinito se c non è un carattere maiuscolo; si dovrà scrivere
if (isupper (c)) c=tolower (c);
a meno che si è del tutto sicuri che è una maiuscola. Solo
if (isascii (c)) if (isupper (c)) c=tolower (c);
è totalmente sicuro.

int **toupper**(c) standard macro ctype.h
TOUPPER ritorna il carattere maiuscolo equivalente al carattere minuscolo c. (Le regole sono le stesse di **tolower**)

int **ungetc**(c,StreamP) standard function stdio.h
char c;

UNGETC scrive ("manda indietro") un carattere nel buffer dell'input StreamP così che il carattere sarà il primo a essere ritornato dalla serie delle chiamate **getc.** EOF non dovrebbe mai essere mandato indietro.
Si dovrebbe aver precedentemente letto qualcosa dallo stream. Un singolo livello di "mandata indietro" è garantito, ma solo se un carattere è stato letto precedentemente dallo StreamP. La funzione ritorna il carattere, o l'EOF se l'operazione non è riuscita.

int **unlink**(Fpath) system function —
Δ VAX

UNLINK rimuove un link dal file specificato. Su sistemi non UNIX ciò è equivalente a cancellare il file; il sistema VAX utilizza invece la funzione **delete.** Se l'operazione è riuscita correttamente si ottiene uno 0, altrimenti un -1. Il file è distrutto solo quando l'ultimo link è stato rimosso.
Potrebbe essere opportuno costruirsi delle proprie funzioni di **delete** e **rename.**

int write(Handle,b,bytes) system function —
char b[100]; Δ KR CC
int bytes=100;

WRITE scrive il numero di byte indicato dal buffer b nel file specificato. Se tutto è corretto la funzione ritorna il numero effettivo di byte scritti. Se sono scritti meno byte di quelli richiesti è segno che c'è stato un errore. In caso di errore si ottiene un -1.

Le Guide di Bit sono uno strumento prezioso per chi lavora con il computer. In poche pagine riassumono con chiarezza quanto è necessario sapere su diversi argomenti di informatica, andando incontro alle più diverse esigenze.

Informatica
Forniscono le conoscenze fondamentali per una cultura informatica di base.
Software
Presentano i pacchetti software e i sistemi operativi più diffusi sul mercato e suggeriscono idee e proposte di programmi per diverse applicazioni.
Linguaggi
Sono comode tabelle di riferimento delle istruzioni e i comandi dei linguaggi più famosi.